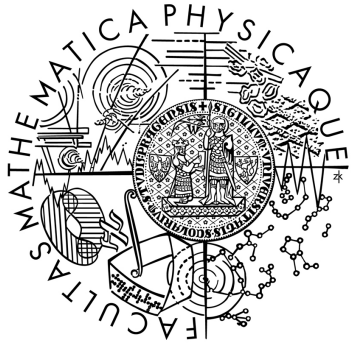
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Tomáš Pošepný

Governmental Linked Data and Experimental Application

Department of Software Engineering

Supervisor of the bachelor thesis: Mgr. Martin Nečaský Ph.D.

Study programme: Computer Science

Specialization: General Computer Science

Prague 2011

My great thanks go to the leader of my bachelor thesis, Martin Nečaský, for inspiring advices and comments and his willingness. I would also thank Jiří Skuhrovec and Pavel Nohejl for giving me the basic legal and economic background for public procurement and answering my numberless questions about SPARQL endpoint respectively.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date 5.8.2011 Tomáš Pošepný

Název práce: Veřejná Linked Data a experimentální aplikace

Autor: Tomáš Pošepný

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Martin Nečaský Ph.D.

Abstrakt: Cílem této práce je představení technologií Resource Description Framework a Linked Data a zmapování současné situace v publikování veřejných zakázek v České republice na Internetu. Práce je zaměřena na problémy a nedostatky zveřejňování a nabízí řešení v podobě Government Linked Data. Současně s tím je vyvinuta experimentální aplikace na sescrapování a triplifikaci veřejných zakázek ze systému E-ZAK s možným rozšířením o další datové zdroje jiných administrátorů. Získaná data v podobě RDF jsou vizualizována spolu s Linked Data datovým zdrojem zakázek z ISVZ, obsahujícím také informace o firmách z ARES systému, v jednoduché webové aplikaci. V závěru je celá práce shrnuta a jsou uvedena některá doporučení pro vývojáře podobných aplikací.

Klíčová slova: Linked Data, RDF, data státní správy

Title: Governmental Linked Data and Experimental Application

Author: Tomáš Pošepný

Department: Department of Software Engineering

Supervisor: Mgr. Martin Nečaský Ph.D.

Abstract: The aim of this thesis is to introduce the technologies of Resource Description Framework and Linked Data and map the current situation in publishing procurement in the Czech Republic on the Internet. The thesis is focused on problems and deficiencies in the publishing and offers solution in the form of Government Linked Data. Along with that, an experimental application for scraping and triplifying public contracts from the E-ZAK system is developed. It also allows possible later extensions for other systems administrating public procurement. The obtained RDF dataset is mashed up with a Linked Data data source for public contracts posted in ISVZ containing also information about organizations from ARES system. The mashed data are visualized in a simple Web application. In the conclusion, the entire thesis is summarized and a couple of hints for developers of similar applications are mentioned.

Keywords: Linked Data, RDF, governmental data

Contents

Introduction **3**

1 Technologies **4**
1.1 Resource Description Framework 4
1.1.1 Introduction to RDF 4
1.1.2 RDF Model 5
1.1.3 Serialization Formats 8
1.1.4 RDF Ontologies 11
1.1.5 RDF Schema 11
1.1.6 Web Ontology Language 13
1.2 Storing RDF Data 14
1.2.1 Native Triple Stores 14
1.2.2 Database Management System - Backed Stores 14
1.2.3 RDF Wrappers 15
1.3 SPARQL 15
1.3.1 Querying RDF Data 15
1.3.2 Introduction to SPARQL 16
1.3.3 Basic SPARQL Queries 16
1.4 RDFa 20
1.4.1 RDFa Attributes 21
1.4.2 RDFa vs. Microformats 22
1.4.3 XHTML+RDFa Documents 22
1.5 Linked Data 23

2 Public Procurement on The Web **26**
2.1 Current State 26
2.2 Weaknesses of Current System 26
2.3 Suggested Solution 27
2.4 Existing Tools For Scraping and Triplification 28

3 Public Contracts Triplification **30**
3.1 Data Source Description 30
3.2 Data Source Analysis 31
3.3 Problem Analysis 31
3.4 Solution Proposal 33
3.5 Implementation 34
3.5.1 Scraping with *jsoup* Library 35
3.5.2 Triplification with *Jena* Framework 37
3.6 Testing 41
3.7 Discussion 42

4 Visualization **44**
4.1 Description 44
4.2 Analysis 44
4.3 Solution Proposal 45

4.4 Implementation 46
4.5 Testing 47
4.6 Discussion 47

Conclusion **50**

Bibliography **52**

Attachments **56**

A CD Content **57**

B User Documentation For Scraper And Triplification **58**
B.1 Graphical User Interface 58

C Programming Documentation For Scraper And Triplification **60**

D User Documentation For Visualization **61**

E Programming Documentation For Visualization **62**
E.1 Used Tools 62
E.2 Documentation 62

Introduction

Back in 1989, Sir Tim Berners-Lee invented the World Wide Web. Something that has moved the world of the Internet into a completely new era. Something that indeed changed the world. Now, the same person has come with an idea of Linked Data - technology that might, in my opinion, change the World Wide Web equally as the Web changed the Internet 22 years ago.

Our lives are about data, pieces of information that we put together, analyze, use to make decisions, understand and solve problems. Data are the source of knowledge. And now, with Linked Data, we can publish the data on the Web, along with the semantics, in a way that they are understood by machines and connected to each other according to the relations between them. It is not about linking documents any more, from now on it is about linking raw data.

Public procurement has been recently one of the most frequently discussed topics on our (not only) political scene. Nobody actually knows how much money is spent on procurement and existing estimates vary up to 150 billion crowns. By the way, this is about what the state deficit is [1]. Transparency is the last word I would use to describe the system of public procurement. I should be careful even with the word system. Currently public contracts are spread across many different web pages that differ also in the structure of published information. In addition, the Information System on Public Contracts which is the central database of public tenders does not contain complete information. Some of it, such as the documentation, is missing. In this thesis I would like to point out some shortcomings of the current system of public procurement on the Internet and mainly focus on the Resource Description Framework (RDF) and Linked Data technologies in conjunction with what benefits these could bring to public procurement, by extension, to government data publishing.

I have set myself several goals I want to achieve in this bachelor thesis:

- to study thoroughly RDF and Linked Data technologies and gain an overview on how Government Linked Data (as an application of Linked Data in the area of government data) was accepted in other countries
- to review the situation of government data on the Web in the Czech Republic with focus on public procurement and show what benefits may be brought by publishing public government data on the Web of Linked Data
- to select one of the procurement administrators, create a scraper and provide the data in RDF
- to create a Web application that mashes triplified data with another Linked Data dataset and shows some simple visualization to prove the ease of use of such data and the potential for many interesting applications

1. Technologies

In this chapter, I would like to provide a basic overview of technologies used in this bachelor thesis. Since most of the technologies are quite new, it should give the theoretical background necessary for the entire thesis, but expects the reader to have basic IT knowledge. For more detailed information, please, refer to related literature and web sources.

1.1 Resource Description Framework

Resource Description Framework (RDF) [2] is a World Wide Web Consortium (W3C) standard whose specification consists of a suite of W3C Recommendations and is described as a language for representing information about resources in the World Wide Web. It was originally created in 1999 on top of XML as a metadata data model. Since then RDF has evolved a lot. If you look at it as a general method for conceptual description and modeling of information implemented in Web resources on the Semantic Web, by generalizing the concept of Web resource to anything that is identifiable, but does not have to be retrieved, on the Web, it has been mainly used to encode information and relations between any material as well as abstract entities. New and so far the latest updated version of Resource Description Framework was published in 2004. [2, 3, 4]

1.1.1 Introduction to RDF

Following description is based on [2, 3, 4, 5].

RDF is intended to be a standard for Semantic Web. Unlike the Web of documents that can be described as a decentralized platform for distributed presentations, the Semantic Web of data is a decentralized platform for distributed knowledge. This "knowledge" or "semantic" basically means that the logic of presented information can be somehow manipulated by a machine and that it is not opaque to it.

The purpose of RDF is to provide a general method to encode information without loss of meaning. The effort is to have that method so simple that any resource can be described in so structured way that it can be understood by any other applications. If we leave the more or less formal definition, RDF is a general way to decompose knowledge into small discrete pieces with some rules about the semantics. These pieces are called RDF statements or triples. But not only that. Considering Resource Description Framework has been developed as a standard for Semantic Web, there are other requirements for it to meet.

Information must be expressed flexibly. Real life entities are changing all the time and if the Semantic Web wants to reflect real life, its standard must provide a way to flexibly react on these changes. Technically speaking, storing data in either hierarchies (eg. XML) or relational databases is not satisfying. Semantic Web information is best expressed as a data graph.

Since entities are related to each other, each of them (relationship is also an entity - abstract one) needs to be identifiable and this creates a problem of ambiguity. There is many things with identical names that we need to distinguish

between. On the other hand, there sometimes exist things with many different names where we need to make sure all of them are actually referring to the same entity in the Semantic Web graph. Therefore RDF has to provide a collection of globally unique identifiers. Not to break the main principle of World Wide Web, these must be able to be assigned in a decentralized way.

RDF can not avoid comparison to XML. Not only is XML the most used serialization format of RDF graph, but it is also designed to be simple, flexible and applicable to any type of data. Despite that fact, they differ quite a lot. One reason is that, as mentioned above, RDF is not a data format. It is more likely a method or model that can be serialized in one of several formats like RDF/XML, Turtle, Notation3, N-Triples or RDFa. The latter main difference is that there is no built-in meaning to an XML file, but RDF is all about conveying semantics and there are standards on top of RDF, including RDF Schema (RDFS) and Web Ontology Language (OWL), that allow logical inferences from data [7].

1.1.2 RDF Model

This section and all its subsections are based on [3, 4, 5].

RDF's underlying structure is called a triple. Any expression in RDF is broken down into a collection of triples what is in formal terminology a labeled, directed graph. Each triple (node-arc-node) represents a statement of a relationship between the two nodes and consists obviously of three parts:

- a **subject** - what the statement is about
- a **predicate** (sometimes called property) - describes the relation between subject and object.
- an **object**

Both concrete (e.g. books, tablets, ...) and abstract (e.g. the state of knowing someone, having something, ...) real-world entities are on the Web represented by resources. Resources are denoted by names and that is what subjects, predicates and objects are. There are three types of names being used in RDF:

- URI references
- literals
- blank nodes.

Whilst a subject might be either URI reference or blank node and a property is always URI reference, an object is allowed to be any of these types.

URI References

Resources that Resource Description Framework uses to represent entities in the world need to be identified by names. These names are then used for subjects, predicates (properties) and objects in RDF statements and must meet some main requirements stemming from the nature of RDF and Semantic Web.

First and obvious criterium for RDF names is to be machine-processable. Otherwise they would be completely useless for the World Wide Web. On top of this, since RDF is designed for the Web, there is tendency to use names that are dereferencable (there is an actual web page behind them).

Another one is to be global. Since RDF is meant to be a standard for distributed knowledge, resources must be identifiable by the same name wherever they are referred to from. So when I use "posepnyBachelorThesis" to be a name for resource denoting this thesis, then whoever wants to refer to my bachelor thesis, can use exactly this name in their application.

Last but not least, the names have to be unique. Using preceding example, no other resource than the one representing this bachelor thesis should be named with "posepnyBachelorThesis".

To meet all the mentioned criteria, RDF uses what is formally known as Uniform Resource Identifiers (URIs). They usually look like URLs which are known from website addresses, but URLs are actually just a particular kind of URIs. The difference is that URIs are not limited to identifying things located on the network. In fact, they can represent absolutely anything, including human beings, companies, business contracts, abstract concepts (such as the state of knowing somebody or being same as something) and, of course, network-accessible resources.

To be more precise, RDF names resources with URI references (sometimes called qualified URIs). Those are URIs with an optional fragment identifier at the end, separated by the "`#`" character. An example of such a URI reference might be `http://purl.org/procurement#Contract` where `http://purl.org/procurement` is the URI part and `Contract` is the fragment identifier. Thus, from RDF's point of view, resource is anything that can be identified by URI reference.

As for the URLs as dereferencable URIs, even thought it is not an obligation, there is a broad agreement that whenever these are used for resources, there actually is some information on the retrieved web page.

One advantage of using URI references I would like to mention is how they help preventing name clashes. This does not necessary need to be true for all URI references, but in most cases, they start with the owner or creator. That means splitting the space of possible names into units by owners. So when a URI reference starts with `http://google.com/`, it is obvious that it is controlled by the Google company and nobody else should use URI references starting like that. However, there is, of course, no supervision over it.

Using the concept of namespaces from XML technology, URI references can be abbreviated. It is a common practice to declare the namespaces at the beginning and then use the abbreviated names with prefixes throughout the RDF document.

Vocabularies To prevent Semantic Web become a mess, there is an effort to use existing URIs as much as possible rather than creating new ones. Not only it helps avoiding many names for the same resources, but, in case of predicates, it also allows to express the same meaning and, as a consequence, make the statement visible to already existing applications. Hence, there exist sets of URI references defined for specific purposes which are called *vocabularies*.

Vocabularies are often organized, so they can be easily used with common

prefixes. One of the mostly used vocabularies are, for example, *Dublin Core* with predicates for describing documents (title, creator, language, ...) or *Friend-of-a-Friend* describing people and links between them (name, nick, knows, ...).

Literals

Instead of URI references, objects of RDF triples might also be literal values. Literal value or simply literals are raw text used to identify values such as people's names, numbers, dates or book titles. They could be represented by a URI reference, but it is a better practise to use literals in these specific situations. Let's say we want to encode information about someones age. Without literals, the triple would look something like:

```
ex:homer_simpson ex:age exnums:40
```

where `exnums:40` is the abbreviated URI reference for number 40. But it can be better encoded with a literal value instead:

```
ex:homer_simpson ex:age ‘‘40’’
```

Extra pieces of information may be appended to literals to specify either the language or datatype. Literals combined with an optional language tag are called *plain* literals, while those with datatype metadata are *typed* literals. The RDF's semantics takes both language tags and datatypes into account. The datatype says how to interpret the text (as a number, date, ...) and the semantics of RDF takes it into account. Although datatypes can be any URI, XML Schema datatypes are used by convention.

Blank Nodes

The last element that can possibly appear in an RDF graph, hence, in an RDF triple is blank or anonymous node. It may only appear as subject or object and is a special node, because it has no global name and cannot be referred to from outside of the particular RDF graph. However, it has a local blank node identifier that allows several local statements to refer to the same anonymous node. These local identifiers have following form: `_:name`

Anonymous nodes serve as intermediate steps, mainly for resources that may not have URI references, but are described by relationships with other resources that do have URI references. One typical example can be to avoid having to assign URI references to people. Instead there is one blank node and statements about it, that are specifying particular properties:

```
_:homer    rdf:type         foaf:Person
_:homer    foaf:givenName   ‘‘Homer’’
_:homer    foaf:familyName  ‘‘Simpson’’
_:homer    foaf:age         ‘‘40’’
```

RDF only allows binary relations, therefore it uses blank nodes as an effective tool for intermediate steps to encode more structured information.

1.1.3 Serialization Formats

So far, I have been talking about the abstract RDF model which is a graph. In order to use RDF in real applications, this graph needs to be serialized. We need to write all those triples down so they can be published, exchanged, queried, visualized and so on. The two most common serialization formats, both introduced by W3C, are RDF/XML and Notation 3 (N3). Even though these formats look completely different, the RDF model behind them remains the same, so it basically does not matter which one is used.

RDF/XML

RDF/XML [6] was introduced among the other W3C specifications defining RDF as an XML format for RDF serialization.

Every RDF graph serialized in RDF/XML must be enclosed in `rdf:RDF` root element which can contain all the namespaces and prefixes declarations being used throughout the document. Namespace with `http://www.w3.org/1999/02/22-rdf-syntax-ns#` URL is mandatory.

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:ex="http://www.example.org />
    ...
</rdf:RDF>
```

Each statement is then represented by `rdf:Description` element with an `rdf:about` attribute giving the URI reference of the resource (or local blank node identifier) it represents. Inside the `rdf:Description` element, there are property elements with information about the property and object of the statement. The basic structure for one statement with object defined by URI reference can be described as follows:

```
<rdf:Description rdf:about="subject">
    <predicate rdf:resource="object"/>
</rdf:Description>
```

When the object is a literal value, the structure looks like this:

```
<rdf:Description rdf:about="subject">
    <predicate>literal</predicate>
</rdf:Description>
```

Statements about same subject can be all written inside one `rdf:Description` element as shown in next example:

```
<rdf:Description
    rdf:about="https://www.example.org/simpsons#Homer">
<rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
<foaf:knows rdf:resource="https://www.example.org/simpsons#Marge"/>
<foaf:age>40</foaf:age>
</rdf:Description>
```

There are three statements encoded in this piece of RDF/XML file. They all have subject with "https://www.example.com/simpsons#Homer" URI reference. In triples, these would be

```
exsimps:Homer rdf:type foaf:Person
exsimps:Homer foaf:knows exsimps:Marge
exsimps:Homer foaf:age "40"
```

Note, that to distinguish between literal values, URI references and abbreviated URI references, a common convention is used. It is to enclose literal values in quotation marks, URI references in angle brackets and leave abbreviated URIs as they are. If there were any anonymous nodes, those would be written similarly to abbreviated URIs, but with "_" as their prefix (e.g. _:bnode1).

To understand RDF/XML files it is important to know, that resources in the role of object might be described in two ways. First one is to put resource's URI reference into the `rdf:resource` attribute and describe the properties in a separate `rdf:Description` element. The latter is to append the `rdf:Description` element as a child of property element and omit the `rdf:resource` attribute.

```
<rdf:Description
       rdf:about="https://www.example.org/simpsons#Homer">
  <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
  <foaf:knows>
   <rdf:Description
       rdf:about="https://www.example.org/simpsons#Marge">
    <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
    <foaf:age>unknown</foaf:age>
   </rdf:Description>
</rdf:Description>
```

So far mentioned basic constructs can be further simplified by so called syntactic sugar. Probably the most used one, that is worth mentioning, is merging `rdf:type` property element and `rdf:Description` resource element. The `rdf:Description` tag is replaced with the type from `rdf:type` element. Instead of writing

```
<rdf:Description
       rdf:about="https://www.example.org/simpsons#Homer">
   <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
</rdf:Description>
```

one can write

```
<foaf:Person rdf:about="https://www.example.org/simpsons#Homer"/>
```

Notation 3

Unlike RDF/XML which is the normative syntax for writing RDF, Notation 3 (N3) [8] is favorite for its readability and ability to clearly capture the RDF graph. It is a shorthand non-XML serialization that is much more human-friendly than RDF/XML in terms of readability. The syntax is pretty much what I have been

using for triples so far. There is a declaration of prefixes at the beginning followed by statements written simply as subject (URI reference, blank node), predicate (URI reference) and object (URI references, blank nodes, literal values) separated by whitespaces and terminated by a period. The `@prefix` directive is used to declare namespaces.

```
@prefix foaf:     <http://xmlns.com/foaf/0.1/> .
@prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix exsimps:  <http://www.example.org/simpsons#> .

exsimps:Homer   rdf:type       foaf:Person .
exsimps:Homer   foaf:knows     exsimps:Marge .
exsimps:Marge   foaf:homepage  <http://simpsons.wikia.com/wiki/
                                          Marge_Simpson> .
```

One of the aims of Notation 3 is to be as readable and natural as possible. Therefore, it provides many syntactical features. The most common ones are [8]:

- using "a" character for `rdf:type`
- separation of more objects for the same subject and predicate by comma
- separation of more predicates for the same subject by semicolon
- blank nodes represented by square brackets with predicate and subject inside them

The following piece of N3 file shows all the above mentioned syntactic sugar (prefixes definitions are omitted).

```
exsimps:Homer   a           foaf:Person ;
                foaf:knows  exsimps:Marge ,
                            exsimps:Bart ,
                            [ foaf:familyName "Flanders" ] .
```

Without the syntactic simplification, this would be serialized as follows.

```
exsimps:Homer   rdf:type        foaf:Person .
exsimps:Homer   foaf:konws      exsimps:Marge .
exsimps:Homer   foaf:knows      exsimps:Bart .
exsimps:Homer   foaf:knows      _:bnode1 .
_bnode1         foaf:familyName "Flanders" .
```

Other Formats

Couple of other serialization formats have been developed that are subsets of one or the other main formats. I would like to mention some of them very briefly.

Turtle Turtle (Terse RDF Triple Language) [9] is a subset of Notation 3 and superset of N-Triples format. [10]

N-Triples N-Triples [11] is a line-based RDF serialization format, subset of the Turtle, that was designed to be a simpler version of Turtle and N3 to allow easier parsing and generating by applications. However, it is missing some significant shortcuts and therefore difficult to read with large amounts of triples. Thanks to its little variation in serializing RDF graph, N-Triples format is used for representing the model answers for parsing RDF/XML test cases. [12]

TriX and TriG TriX (Triples in XML) [13] is another XML-based serialization format for RDF graphs. It was developed as a simple solution to known problems in RDF/XML and added an ability of describing named graphs [14]. TriG [15] is a compact and readable non-XML alternative to TriX [16].

1.1.4 RDF Ontologies

Besides statements about relationships between real-world entities, RDF can be used at a higher level to define also the terms used in those statements. It can describe classes and properties of those resources in so called *ontologies*.

RDF ontologies are the true spirit of RDF. It is them that enrich RDF data models with the semantics by indicating how some information should be interpreted and a new one inferred. In other words, RDF ontologies are metadata about RDF (written in RDF). There are two principal syntaxes for ontologies, RDF Schema (RDFS) and Web Ontology Language (OWL), both of which are W3C specifications.

1.1.5 RDF Schema

Description of RDF Schema is based on [17, 18, 3].

RDF Schema [17], also known as RDF's vocabulary description language, provides facilities to describe classes and properties of RDF resources. These facilities are classes and properties themselves. In other words, RDF Schema is a vocabulary for describing other vocabularies used in RDF statements and graphs. To sum it up, RDF Schema are valid RDF statements with some reserved terms which there is an agreement about meaning for.

It is built on the limited vocabulary of core RDF whose namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#` I will use the common `rdf` prefix to refer to. RDF Schema itself is defined in `http://www.w3.org/2000/01/rdf-schema#` namespace which is usually abbreviated as `rdfs`.

Following paragraphs will introduce the most common classes and properties used to describe classes and properties, together with real examples. However, it is not to be considered a complete list of RDF Schema members. Also note, that examples are written in N3 triples, the declarations of namespaces are omitted and common prefixes are used.

Describing Classes

In RDF, classes are thought of as kinds of things or categories. In addition, all classes are subclasses of `rdfs:Resource` class. That is the class of all resources. Another important class is `rdfs:Class`. All classes are instances of this class. To explicitly denote that a resource is an instance of a class, `rdf:type` predicate

is used. Here are some examples, how classes are defined (note, that `rdfs:Class` is an instance of its own):

```
rdfs:Class      rdf:type   rdfs:Class .
foaf:Person     rdf:type   rdfs:Class .
exsimps:Homer   rdf:type   rdfs:Person .
```

To define hierarchies in classes, `rdfs:subClassOf` property is used. Next triple expresses, that every instance of `foaf:Person` is also an instance of `foaf:Agent`.

```
foaf:Person   rdfs:subClassOf   foaf:Agent .
```

Describing Properties

To mark a resource as a property it must be an instance of `rdf:Property` class. This is how new property with `ex:likes` URI reference would be explicitly defined:

```
ex:likes   rdf:type   rdf:Property .
```

To further describe the property, there are two extra properties in RDFS:

- `rdfs:domain`
- `rdfs:range`

They specify the class for subject or object of the relevant property, respectively. I think the names in RDF Schema are all pretty self-explanatory, but here is an example:

```
ex:likes   rdfs:domain   foaf:Person .
ex:likes   rdfs:range    rdfs:Literal .
```

To translate these triples into sentences, subjects of `ex:likes` property are instances of `foaf:Person` class, while objects are instances of `rdfs:Literal` class which defines literal values. These two properties are more important than it may seem. They are not only helpful as a documentation for other people to help them use the vocabulary properly, but also allow applications to make inferences. Since I consider this an important feature, I will discuss it in a separate paragraph.

There is also an alternative to `rdfs:subClassOf` for properties. Two resources related by a property that is defined as `rdfs:subPropertyOf` some other property, are, as a consequence, related also by that other property.

Those were just the most frequent facilities. Other classes and properties are covered in RDF Schema, but I do not think listing them all would make any benefit, because they can be found in the W3C specification. I would rather write a few lines about what separates RDF Schema from other, conventional and widely known models.

Inferences

As opposed to classic models (XML Schema, UML, Entity-Relation), RDFS considers metadata as additional descriptions of resources, not constraints. And it leaves it up to the applications to decide how these descriptions will be treated. And this is where the semantics is hidden, since an application can use the metadata to infer new or missing facts, that are implicitly not given. Considering following statements:

```
foaf:knows   rdfs:domain          foaf:Person .
foaf:knows   rdfs:range           foaf:Person .
ex:likes     rdfs:subPropertyOf   foaf:knows .
```

and

```
exsimps:Homer   rdf:likes   exsimps:Marge.
```

other statements can be inferred:

```
exsimps:Homer   rdf:type    foaf:Person .
exsimps:Marge   rdf:type    foaf:Person .
exsimps:Homer   rdf:knows   exsimps:Marge .
```

Besides inferring new logical consequences, there are some other advantages. As a result of property definitions being, by default, independent of class definitions, properties can be described without domain being specified. That allows to use properties globally and with any class. Such properties can be then used even in applications they were originally not designed for.

At the same time, all this must be used with care not to become disadvantages. For instance, using some properties inappropriately may lead to wrong inferences:

```
ex:likes   rdf:type    rdf:Property .
ex:likes   rdf:range   foaf:Person .

exsimps:Homer   ex:likes   exsimps:Marge .
exsimps:Homer   ex:likes   exfood:Donut .
```

allows to infer:

```
exfood:Donut   rdf:type   foaf:Person .
```

1.1.6 Web Ontology Language

Since the expressiveness of RDFS is limited, Web Ontology Language [19] was created. OWL 2 Web Ontology Language [20], which is the current version of OWL W3C Recommendation, adds more vocabulary for describing classes and properties and is much richer language than RDFS, but the purpose remains the same - making the Web content more and more machine-processable.

Among many features, OWL 2 includes cardinality restrictions, symmetric, transitive or functional properties, distinguishes between datatype and object properties, provides logical class constructors (intersection, union and complement). But it is important to keep in mind, that it is still up to the applications if they use them and how.

1.2 Storing RDF Data

Resources used for this entire section are [21, 22, 23].

As a consequence of RDF breaking knowledge down into triples, even simple information may require quite a lot RDF statements to be encoded. And while small RDF graphs like RDF schemas can be efficiently manipulated in computer's memory, larger ones require a persistent storage. Such a storage could be simply, for instance, RDF/XML file, but for large amounts of triples, database management systems are more suitable. RDF store, also called RDF database or triple store, is defined as purpose-built database for the storage and retrieval of any kind of data expressed in RDF.

Some triple stores can contain even billions of triples. A benchmark was developed on the Lehigh University[1] to measure the performance of triple stores. [24]

Depending on the architecture, three types of triple stores can be distinguished: *native triple stores*, *DBMS-backed stores* and *RDF wrappers*.

1.2.1 Native Triple Stores

Native triple stores are built primarily for RDF and implement their own complete database engine from scratch. As opposed to DBMS-backed stores, they are independent of any other database management system. Some examples of native stores are Jena TDB, Mulgara or Talis Platform. According to the Europeana RDF store report [21], it seems that RDF store development goes right into the direction of native stores.

1.2.2 Database Management System - Backed Stores

DBMS-backed stores, such as 3store, ARC or Boca, represent completely opposite principle. These are stores built on top of existing database engines. Mostly, DBMS-backed stores make use of relational database management systems (RDBMS). Such an approach provides an off-the-shelf solution and building new powerful engines costs just little extra programming.

Probably the biggest challenge here is how to map RDF graph to database tables. Many different schemas exist that can be split into two groups: *generic schemas* and *ontology specific schemas*.

Generic Schemas

Generic schemas are not aware of the ontology of RDF data. Therefrom resulting advantage is, that no restructuring is required when the ontology changes. However, every query searches the whole database and non-trivial queries cause number of joins which are expensive.

The most simple generic schema maps RDF statements to a single database table with three columns reflecting the structure of an RDF triple.

[1] `http://swat.cse.lehigh.edu/projects/lubm/`

Ontology Specific Schemas

RDF relational databases using an ontology specific schema consider the ontologies when creating tables. Whenever the ontology changes it is propagated to the database tables. The basic ontology specific schema contains one table with columns for instance ID, class name and each property. Hence, each row represents one instance. Because of its obvious shortcomings (large number of columns, new properties in ontology cause adding new columns etc.), this schema is usually further modified.

One modification, that can be thought of as a horizontal splitting of the basic schema, is known as *one-table-per-class schema*. There is an individual table for each class in ontology, holding all its instances. The class table only contains columns for properties domain set to this class. Such a design allows fast queries about all properties of an instance, however, the problem of changes in the ontology causing tables' restructuring remains.

Another approach is to split the basic schema vertically. This schema, called *one-table-per-property* schema (or *decomposition storage model*), stores one table for each property, including properties from RDF Schema and core RDF. The property table has just two columns, one for the subject and one for the object. The weak spot of this schema are too many joins when queries contain many properties.

The advantages of both are combined in *hybrid schemas*. One table per class is created, storing IDs of instances. This substitutes the `rdf:type` property. Other properties are modelled as in the *one-table-per-property* schema. Thus, changes to the ontology result in creating new tables instead of restructuring of the existing ones.

1.2.3 RDF Wrappers

A little bit different approach is presented by RDF wrappers. An RDF wrapper is a software component that exposes data stored in some existing data source as RDF without any effects to the original data. Accordingly, RDF wrappers offer read-only access. The data sources that they can be set up on top of, are, for example, relational databases, structured file formats or complete file systems.

1.3 SPARQL

To actually use data stored in triple stores, there must be a way of interacting with them. The most common one is the use of query languages. Probably the best known example is Standard Query Language (SQL) for relational databases. For RDF stores, its role is taken by SPARQL Protocol and RDF Query Language (SPARQL) [25].

1.3.1 Querying RDF Data

When querying RDF data, three different levels of abstraction can be considered: *syntactic level*, *structure level* and *semantic level*.

Following description is based on [22].

RDF data serialized in RDF/XML format are, at the syntactic level, XML documents. Thus, they could be queried as XML documents by using some of the XML query languages (eg. XPath or XQuery). However, the significant differences between RDF and XML technologies are the reason for several drawbacks (mainly querying the relationships that are explicitly not written) of this approach.

A higher level of abstraction interprets RDF data as triples, but without the semantics. The so called transitive closure is not provided which means that the transitive relations (inferences) are ignored.

Finally, at the semantic level, queries are based on the graph representation of the data. Hence, not only explicitly represented facts, but also the semantic inferences are considered. This is achieved either by storing the deductive closure of a graph, or by the query processor, but there is no inference it the query language itself.

1.3.2 Introduction to SPARQL

SPARQL [25], which is a W3C Recommendation, is a query language designed for RDF, hence the Semantic Web. To be more precise, it is a set of three W3C Recommendations: SPARQL Query Language for RDF [25], SPARQL Protocol for RDF [26] and SPARQL Query Results XML Format [27]. So it is not only a query language, but also a data access protocol for querying RDF stores remotely and closely related XML format for the results of SPARQL SELECT and ASK. [28]

No doubt, SPARQL query language is the predominant one. It is based on graph patterns matching and enables queries over distributed data sources on the Web. Hereby, it overcomes any other querying language. Since it only is capable of asking queries, SPARQL Update [29] was developed as a companion language.

To expose data from a triple store, SPARQL provides so called SPARQL endpoint where queries can be asked.

1.3.3 Basic SPARQL Queries

The text is based on [30, 31, 32].

SPARQL defines four types of queries:

- *SELECT* - similar to SQL's SELECT, returns matched data in a tabular form
- *ASK* - returns boolean value whether there are any matches or not
- *CONSTRUCT* - results are transformed into RDF triples
- *DESCRIBE* - returns RDF graph with triples somehow related to the matched resources

The keyword for query type is followed by a sequence of variables to be returned. Variables in SPARQL are prefixed by either "?" or "$" and have a global scope. An asterisk character can be used as wildcard to output all variables.

Next part of SPARQL query is optional and specifies the dataset(s). The keywords here are FROM and FROM NAMED for default and named graphs, respectively. The default graph is optionally created by merging all graphs from FROM clauses. In addition, zero or more named graphs might be queried.

WHERE clause is where the graph pattern goes. It is expressed in Notation 3 format, except that also the variables are permitted. The graph pattern may contain abbreviated URI references, if the prefixes are declared at the beginning of the query with the PREFIX keyword.

For examples on SPARQL queries, I will use following RDF dataset as default.

```
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix exsimps:<http://www.example.org/simpsons#> .

exsimps:Homer   foaf:knows        exsimps:Marge ;
                foaf:givenName    "Homer" ;
                foaf:familyName   "Simpson" ;
                foaf:age          40 .
exsimps:Marge   foaf:givenName    "Marge" ;
                foaf:familyName   "Simpson" ;
                foaf:age          36 .
exsimps:Bart    foaf:givenName    "Bart" ;
                foaf:familyName   "Simpson" .
exsimps:Lisa    foaf:firstName    "Lisa" ;
                foaf:familyName   "Simpson" .
```

Here is a simple SPARQL query that returns given name for anybody whose family name is Simpson.

```
PREFIX exsimps: <http://www.example.org/simpsons#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
        ?somebody   foaf:familyName   "Simpson" ;
                    foaf:givenName    ?name .
      }
```

Results:

```
-----------------
|     name      |
=================
|    "Homer"    |
-----------------
|    "Marge"    |
-----------------
|    "Bart"     |
-----------------
```

Another useful keyword is FILTER. It can be used inside the WHERE clause to constrain the value of a variable. I modified the previous example to only return given name of those older then 36.

```
PREFIX exsimps: <http://www.example.org/simpsons#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
        ?somebody   foaf:familyName   "Simpson" ;
                    foaf:givenName    ?name ;
                    foaf:age          ?age .
        FILTER (?age > 36)
      }
```

Results:

```
-----------------
|     name      |
=================
|    "Homer"    |
-----------------
```

SPARQL also offers a way not to reject the solution when part of the pattern does not match. The optional part is specified with the OPTIONAL keyword. To demonstrate this, I wrote a query to return the given name of every Simpson together with their age, if available.

```
PREFIX exsimps: <http://www.example.org/simpsons#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
        ?somebody   foaf:familyName   "Simpson" ;
                    foaf:givenName    ?name .
        OPTIONAL {
                   ?somebody   foaf:age  ?age .
                 }
      }
```

Results:

```
------------------------
|     name      |  age  |
========================
|    "Homer"    |  40   |
------------------------
|    "Marge"    |  36   |
------------------------
|    "Bart"     |       |
------------------------
```

When there is more possible alternatives and it does not matter which one has to match in order to return it as solution, UNION is placed between these alternative patterns. If both branches of the UNION match then both solutions will be returned. It is best seen on a concrete example.

```
PREFIX exsimps: <http://www.example.org/simpsons#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
        ?somebody   foaf:familyName   "Simpson" ;
        { ?somebody   foaf:givenName   ?name }
        UNION
        { ?somebody   foaf:firstName   ?name}
      }
```

Again, the query returns the given name for all Simpsons, but now it allows the name to be provided also as the `foaf:firstName` property:

```
-----------------
|     name      |
=================
|    "Homer"    |
-----------------
|    "Marge"    |
-----------------
|    "Bart"     |
-----------------
|    "Lisa"     |
-----------------
```

So far, the FROM clause has been omitted, because the queries have been asked against default dataset. This dataset can be extended using the FROM keyword with graph's URI. Moreover, multiple named graphs can be added with the FROM NAMED construction. Within the graph patterns, the named graphs are queried with the GRAPH keyword.

Assuming that the RDF data about the Simpson family is stored in simpsons.rdf, the SPARQL query could be as follows:

```
PREFIX exsimps: <http://www.example.org/simpsons#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
FROM NAMED <simpsons.rdf>
WHERE {
        GRAPH <simpsons.rdf> {
            ?somebody   foaf:familyName  "Simpson" ;
                        foaf:givenName    ?name .
            }
      }
```

Results are, of course, same as if the data was in the default dataset:

```
-----------------
|     name      |
=================
|    "Homer"    |
```

```
-----------------
|    "Marge"    |
-----------------
|    "Bart"     |
-----------------
```

Many further refines of the results are possible in SPARQL. DISTINCT keyword right after the SELECT causes only unique solutions to be returned, LIMIT and OFFSET keywords placed after the WHERE clause limit the number of solutions and specify the start index for solutions from all generated ones, respectively. To sort the results, ORDER BY can be used with an optional ASC() or DESC() modifier. Following query sorts lexically the names from previous example and returns only the first two.

```
PREFIX exsimps: <http://www.example.org/simpsons#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
        ?somebody   foaf:familyName   "Simpson" .
        { ?somebody   foaf:givenName   ?name }
        UNION
        { ?somebody   foaf:firstName   ?name}
      }
ORDER BY ASC(?name)
LIMIT 2
```

Results:

```
-----------------
|     name      |
=================
|    "Bart"     |
-----------------
|    "Homer"    |
-----------------
```

A bunch of new features have been added to SPARQL 1.1 [33] which is still under development. On 12 May 2011 the Last Call Working Draft was published, therefore, reaching the Recommendation status is expected pretty soon.

SPARQL 1.1 introduces many new possible constructs. Among others, subqueries, existence filters, aggregates, property paths or federated queries have been added. [33]

1.4 RDFa

Before RDF, the data had only been intended for human consumption. With the advent of RDF, it started being enriched with semantics, with machine-readable metadata. However, this has been done separately and hence with duplicate data. There has usually been a presentation part consisting of an HTML or XHTML

document and a metadata part publishing the actual RDF data either as a file in some serialization format (RDF/XML, N3 etc.) or stored in a triple store. While such a separation does not matter to applications, web browsers only see the presentation part and can not make any use of the knowledge in RDF data. Resource Description Framework - in - attributes (RDFa) [34] combines both segments by providing a thin layer of markup to embed RDF statements within XHTML pages. [35, 36]

That enables the advantages of RDF to be brought closer to an average Internet user, because the browsers are getting smarter and more helpful. For instance, if somebody publishes info about an event or a place using RDFa, the web browsers can allow anybody visiting that page to simply add that event to their calendar or get directions to that place respectively. In addition, there is no need for duplicate data any more.

1.4.1 RDFa Attributes

In this subsection, I used following sources [34, 37].

Technically, RDFa is a W3C Recommendation that specifies a set of few simple XHTML attributes. More precisely, RDFa makes use of a number of XHTML attributes and adds couple of new ones.

The latest version is RDFa 1.1 [38] which is currently the Last Call Working Draft and is intended to become a W3C Recommendation.

Relevant XHTML attributes:

attribute	possible content	meaning
@rel	list of CURIEs	predicates if objects are resources; objects are in @href or @src
@rev	list of CURIEs	reverse predicates if objects are resources; objects are in @href or @src
@content	string	machine-readable content for literal values when using @property
@href	URI	resource of an object
@src	URI	resource of an embedded object

Attributes added by RDFa:

attribute	possible content	meaning
@about	URI or SafeCURIE	subject (what the data is about)
@property	list of CURIEs	predicate if object is literal
@resource	URI or SafeCURIE	resource of an object that is not intended to be "clickable"
@datatype	CURIE	datatype of a literal
@typeof	list of CURIEs	RDF types of subjects

New attributes in RDFa 1.1 [38]:

attribute	possible content	meaning
@prefix	list of prefix-name URI pairs	declaration of prefixes
@profile	list of URIs	profiles' references
@vocab	URI	default mapping if none prefix used

Note, that RDFa allows compact URIs (CURIEs) for resources. CURIE is an abbreviated URI that uses previously declared prefix (eg. `dc:title`). SafeCURIE is a CURIE enclosed in square brackets (`[dc:title]`).

1.4.2 RDFa vs. Microformats

RDFa might seem similar to microformats. They both put pieces of metadata into XHTML documents, but RDFa goes further. While microformats specifies the exact set of terms, RDFa is just about the structure of the metadata. It comes with a fixed set of attributes and a specified syntax, but that is about it. Any RDF vocabularies can be used inside these attributes making RDFa highly flexible. Some of the microformats have been mapped to RDF vocabularies. [34]

1.4.3 XHTML+RDFa Documents

RDFa is designed for XHTML, because HTML language is not extensible. However, it can also be used with HTML and easily imported into other XML-based language. XHTML+RDFa 1.0 [34] is the official document type, but XHTML+RDFa 1.1 [39] is about to gain the status of W3C Recommendation. In this subsection I would like to provide a simple example of XHTML+RDFa document to demonstrate the use of RDFa.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.1//EN"
   "http://www.w3.org/MarkUp/DTD/xhtml-rdfa-2.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" version="XHTML+RDFa 1.1"
      lang="en"
      xml:lang="en"
      xmlns:dc="http://purl.org/dc/terms/"
      xmlns:foaf="http://xmlns.com/foaf/0.1/"
      xmlns:exsimps="http://www.example.org/simpsons#"
      prefix="dc: http://purl.org/dc/terms/
              foaf: http://xmlns.com/foaf/0.1/
              exsimps: http://www.example.org/simpsons#">
 <head>
  <meta http-equiv="content-type" content="application/xhtml+xml;
       charset=UTF-8"/>
  <meta property="dc:title" content="Homer's web page"/>
  <meta rel="dc:creator" resource="[exsimps:Homer]" />
 </head>
```

```
 <body>
  <div about="[exsimps:Homer]" typeof="foaf:Person">
   <p>Given Name:
      <span property="foaf:givenName">Homer</span>
   </p>
   <p>Family Name:
      <span property="foaf:familyName">Simpson</span></p>
   <p>Email: <a rel="foaf:mbox" href="mailto:homer@simpson.com">
              homer@simpson.com
             </a>
   </p>
   <p>Knows:
      <span rel="foaf:knows" resource="[exsimps:Marge]">
         Marge Simpson
      </span>
   </p>
  </div>
 </body>
</html>
```

For compatibility with both XHTML+RDFa 1.0 and XHTML+RDFa 1.1 namespaces are declared both ways, using XML namespaces and `prefix` attribute. Inside the `head` element, document's title and creator are encoded. The title is raw text, whilst creator is identified by SafeCURIE of their resource. The `div` element encodes five RDF statements about a resource specified in its `about` attribute. Here are the RDF triples extracted from the above document in Turtle format:

```
@prefix dc: <http://purl.org/dc/terms/> .
@prefix exsimps: <http://www.example.org/simpsons#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<url> dc:creator exsimps:Homer;
      dc:title "Homer's web page"@en .

exsimps:Homer a foaf:Person;
    foaf:familyName "Simpson"@en;
    foaf:givenName "Homer"@en;
    foaf:knows exsimps:Marge;
    foaf:mbox <mailto:homer@simpson.com> .
```

where `url` is the URL of the document.

1.5 Linked Data

Linked Data technology [45] is another great idea of the inventor of the World Wide Web, Tim Berners-Lee. It is a next level of the Semantic Web (aka Web of Data). With Linked Data, the Semantic Web is not only about publishing data with semantics any more, it is about making links between those data. That

enables navigation between different data sources all across the Semantic Web. The idea itself is very similar to the idea of the current Web, which is sometimes called the Web of Documents. However, whereas the Web is built on the principle of linking HTML documents, Linked Data is more about interlinking individual pieces of data inside these documents.

The value of the Semantic Web is in that it exposes data along with the machine-readable semantics. Nevertheless, in the real world nothing exists separately. Thus, if the aim of the Semantic Web is to encode real-world entities so that computers can understand them as well as humans, it must reflect also the connections. With the links, data become more discoverable, consequently more valuable and useful.

The underlying technology for Linked Data is RDF. To connect two resources from different datasets and define a new link between them, a simple RDF statement is made. Such statements are called RDF links.

Tim Berners-Lee, in his note on Linked Data, outlined four basic principles [45]:

1. *Use URIs as names for things.*
2. *Use HTTP URIs so that people can look up those names.*
3. *When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL).*
4. *Include links to other URIs, so that they can discover more things*

This basically means using RDF with some specific rules. First one is mentioned in the second principle and that is using URLs instead of more general URI references. URLs are dereferencable URIs. In other words, resources, or some information about them, can be retrieved over the HTTP protocol. Second restriction is a duty to include a link to some other resource to express that these two are somehow related.

Recently, data from many different industries have been published as Linked Data and there has been also related increase of applications consuming them. Data about Linked Data datasets are maintained by Linking Open Data community[2] on CKAN. These data are visualized in the Linking Open Data cloud diagram [40] maintained by Richard Cygniak and Anja Jentzsch.

[2] `http://ckan.net/group/lodcloud`

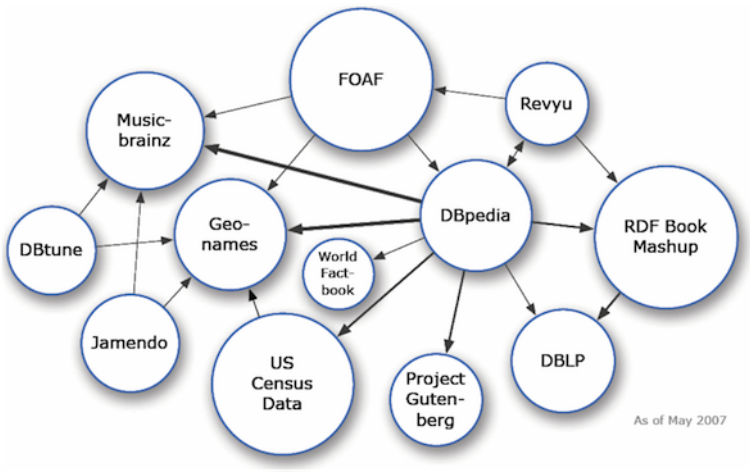


Figure 1.1: First Linking Open Data cloud diagram from May 2007

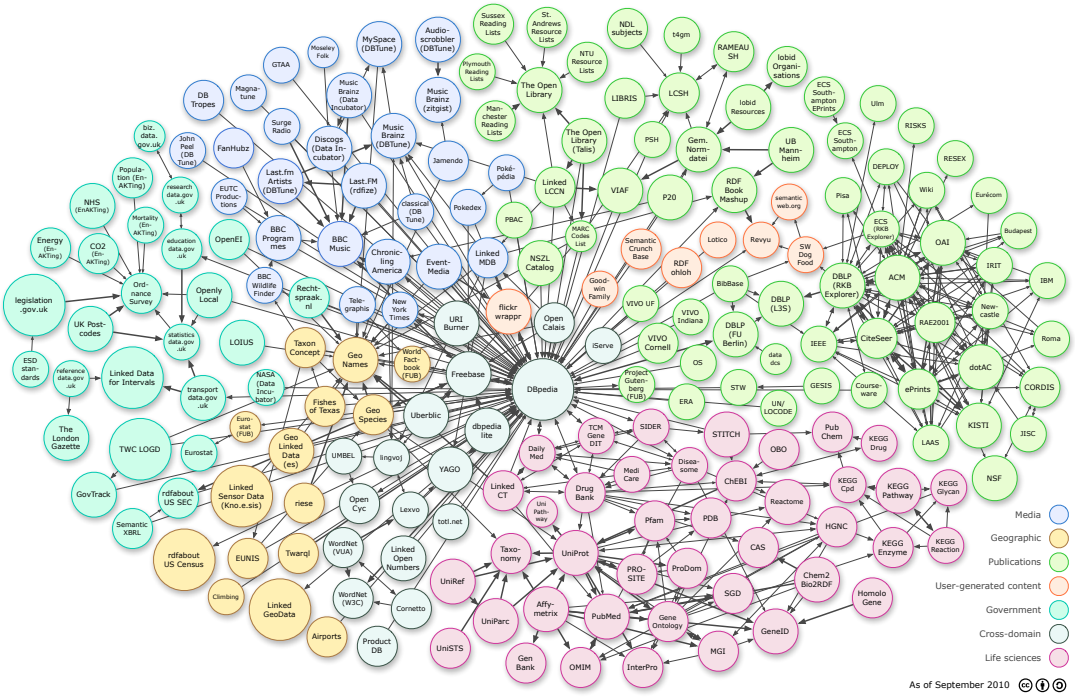


Figure 1.2: Latest Linking Open Data cloud diagram from Semptember 2010

2. Public Procurement on The Web

In this chapter I would like to focus on how public contracts are currently published on the Web in the Czech Republic. I will try to review the main practices and systems as well as their shortcomings and limitations. Along with that, I will introduce possible improvements, by extension, solutions.

2.1 Current State

Recently, there has been a lot of talking about public procurement. In these, from economical point of view, hard times, when the government has declared budget deficit reduction and fight against corruption, we can hear speculations about the amount of money spent in public tenders from all sides. The most frequent number is 600 billion crowns. Accordingly, calls for more transparent procurements are reasonable. In my opinion, major part of this problem is associated with publishing the information on the Internet.

According to bill, submitters are currently obliged to publish tenders with the anticipated price above 2 million crowns without tax for supplies and services and 6 million crowns for constructions. These contracts have to be posted in the publishing subsystem of the Information System on Public Contracts (ISVZ) [41] on the official site of public contracts maintained by Czech Post, by extension, by the Ministry for Regional Development. Public contracts up to these limits are called small amount public contracts and do not fall into the Act on Public Procurement, hence do not have to appear on the Internet. However, they still need to observe the general principle of transparency.

The common practice is that there are many certified systems for procurement offering their services to submitters. Submitters publish their tenders through these systems either on their own website, or on the website of that particular system. Besides that, the system takes care of publishing in the ISVZ. Another approach is that submitters can have their own system and/or they simply publish the contracts directly through the ISVZ. Finally, some smaller submitters that rarely deal with procurement over the mentioned limits usually just use the ISVZ and nothing else.

2.2 Weaknesses of Current System

Even though such a structure does not necessarily need to look bad at a glance, there is many problems and drawbacks in it.

Firstly, there are the small amount public contracts that do not have to be published at all. Even though most submitters report them, there are still many, such as small towns, that do not. Moreover, they are not included in ISVZ and so it is practically impossible to get info about all of them for further potential research, reuse or statistics.

An opposite problem is the data being duplicated. As for the contracts compulsorily published in the ISVZ, they are mostly also posted somewhere on the submitter's web page and thus the data must be maintained at two different places. On the other hand, there are usually couple of documents, such as the documentation, attached to the public contract, but none of them appear in ISVZ.

Although it might sound strange or maybe funny, some public contracts are disappearing from the Web. The biggest procurement administrator is GORDION, s.r.o.[1] Many important submitters publish their tenders through this system, among them, for instance, FN Motol, Ministry of Transport, Finance or Defence. However, on the website where GORDION publishes all the contracts, there is just about 120 latest items and about the same number is in the archive. Everything else is hidden on the private servers running on clients' intranets. In addition, to get permanent link for a concrete contract, you must click a link after which a box with the permanent link appears. This is how transparency is understood by the major procurement administrator in the Czech Republic.

To make a long story short, the data about public contracts, if available, are spread across many systems and web sites and even the central catalogue - ISVZ - does not contain complete information. However, the technical problem that I want to point out, the problem that, in my opinion, ultimately prevents the data about public contracts from being actually useful, more transparent, easily interpretable as well as understandable, the problem that blocks better insight, is the format in which it is being published.

Most often the data are published as PDF documents or there is an HTML table with basic information and couple of PDF files attached. That is completely fine as long as people are just looking at these data, because these formats are intended to be human-readable. However, it becomes useless as soon as somebody would like to use them in their application, to combine them with some other data, to provide another view on them, simply because they are anything but machine-readable.

2.3 Suggested Solution

The solution in four words is Linked Open Government Data. The idea of Open Government Data [42] is very simple and means the disclosure of all government data so it can be *freely used, reused and redistributed by anyone* [43]. There are different levels of openness corresponding to Tim Berners-Lee's 5 star data [45] and Linked Data is the top one.

Many open government data initiatives have been formed around the world and working hard to explain their politicians and public administrations the idea of open data. This has led to governments actually starting opening their data by posting the datasets on the Internet. The pioneers, US[2] and UK[3] governments, have proved it as a right way and have recently been followed by other countries, such as New Zeland or Norway.

Opening up the data changes radically the relationship between people and the government and brings benefits for both sides. People are suddenly much

[1] http://www.gordion.cz/

[2] `http://www.data.gov/`

[3] `http://data.gov.uk/`

more initiated into what is happening. Developers can use the data and add a new value to them, show new views in different contexts and create many different visualization so people can better understand those data. Government data are a breeding ground for many useful applications that can make people's lives easier. Example of such an application can be found in Vancouver where people can register online to be reminded the garbage day. In the United Kingdom, Where Does My Money Go?[4] is a nice example of visualization where anybody can check how the British government spends their taxes. Some other stories can be found at `http://opendatastories.org` and definitely worth watching is also the Open Government Data Film [44]. As a result, opening up the data can lead to economical growth.

The best way to open government data is by using Linked Data. The real pioneer here is the UK government who has committed that it would make any raw dataset available as Linked Data. Using Linked Data standards for publishing government data has many advantages. Namely this can be, for example, modularity and responsibility of publishing, because the publisher is in control of their data by determining what information is returned when URLs are dereferenced. [46]

Finally, everything that has been said can be applied to the public procurement with the same consequences, advantages and potential of more transparent system generating economic value. Technically, the way from publishing data on the Web to publishing them in an RDF format is not that long, thanks to RDFa.

In the Czech Republic, it is the OpenData.cz [47] initiative who is trying to persuade the public and state administration to this approach. They have even created an RDF ontology for public procurement[5] which was an important step since there was not any that could be used. That basically means determination of basic shared template for procurement, because the law does not speak about what information should be published. At the same time, as follows from the nature of RDF, this is just a recommendation setting the minimum and does not restrict the submitters to post any extra information. On top of that, OpenData.cz have introduced a complete tutorial [48] on marking (X)HTML documents with public contracts using RDFa and their ontology. That makes it really easy to switch to Linked Open Government Data and at almost no costs.

To sum it all up, modern web technologies headed by RDF and Linked Data represent an easy and beneficial way that could shift procurement to next level. The technical background has been prepared, its contribution proved by pioneers of Linked Open Government data and OpenData.cz has made it even easier by publishing a tutorial and developing a specific ontology. Now it is submitters', public administration's and government's turn.

2.4 Existing Tools For Scraping and Triplification

Several handy tools for scraping and data triplification have been developed. As for the scraping, it is probably impossible to develop a tool that could be reused for

[4] `http://wheredoesmymoneygo.org/`

[5] The ontology can be downloaded in RDF/XML or Turtle format at [48]

any page. Scrapers are usually designed for a concrete web pages and thus useless for others. What we can reuse are the libraries for parsing HTML. However, instead of providing an overview of many libraries for different programming languages, I would like to highlight a little bit different tool - ScraperWiki.

ScraperWiki [49] is a platform that allows anybody to create or request a scraper. The good thing about it is that everything is set up and ready to use and the scripts as well as scraped data are kept on the website. The data can be then downloaded as a CSV file or accessed through an API. ScraperWiki also offers the option of running the scripts automatically according to a schedule. Moreover, a lot of scrapers are public so anybody can use them, learn from them or modify them. At the moment, one of three languages can be used, including PHP, Ruby and Python. Besides creating scrapers, ScraperWiki also allows people who are not that skilled to write a scraper themselves to request data either from the developers community (for free) or from the ScraperWiki team as a paid service.

The situation with triplification tools is much better. Still more and more tools are being developed to convert some application-specific data into RDF data. Most work can be seen on tools triplifying relational databases. That is reasonable, because most data now is stored in SQL databases. Some examples worth mentioning are *D2R Server* or *Triplify*.

D2R Server [50] is a tool for publishing relational databases as RDF data. It provides a language to define mappings between the original database and an RDF ontology. This mapping is then used to provide RDF or even Linked Data view over the database and allows to query the data using SPARQL protocol, Jena or Sesame APIs. [50, 52]

Triplify [51] is a small plug-in for existing Web applications. It makes the relational database available as Linked Data, by extension, RDF and JSON. Premade configurations exist for Drupal, Joomla, phpBB, WordPress and others. [51, 53]

However, not only relation databases can be triplified. There are many tools for other formats as well, including Excel tables, JPEG or iCalendar. Continuously updated list of converters to RDF can be found at [54].

3. Public Contracts Triplification

One of the aims of this bachelor thesis is to provide an experimental application to show the advantages of RDF and Linked Data. In this chapter I will introduce the first step - scraping the data about public procurement and their triplification. I will built a simple application that scrapes data with plugged-in scraper and exports them in one of the RDF serialization formats.

3.1 Data Source Description

First of all, I had to choose a data source to scrape. The Information System on Public Contracts (ISVZ) has one undeniable advantage that it contains items from all other systems. However, as I mentioned, there are no small amount public contracts there and the others are missing originally attached documents. Therefore I preferred one of the specialized systems, one of the public contracts administrators that submitters publish their tenders through before these are sent to ISVZ. Since I wanted to get a decent number of data from different, possibly interesting, submitters, I finally decided to go with E-ZAK[1].

E-ZAK is one of the major electronic systems for the administration of public contracts. It supports all kinds of public contracts including small amount public contracts. At the present time, E-ZAK services are used by a number of submitters:

- Ministry of Regional Development CZ
- State Institute for Drug Control
- Masaryk Memorial Cancer Institute
- St. Anne's University Hospital Brno
- Masaryk University Brno
- Palacký University Olomouc
- Jan Evangelista Purkyně University in Ústí nad Labem
- University of Ostrava
- Statutory City of Liberec
- Town of Dvůr Králové nad Labem
- Central shopping (Pilsen Region)
- CESNET
- CEJIZA

[1] `http://www.ezak.cz/verejne-zakazky`

As for the architecture, E-ZAK is a client-server application. It does not post all the public contracts in one place, but separately for each submitter on their website. On top of that, there is a web page[2] with contracts of different, usually smaller submitters that do not have their own E-ZAK web page.

3.2 Data Source Analysis

E-ZAK publishes information in the form of HTML documents. Each submitter has their own page with list of contracts. This list contains just basic info about contract, namely the name, limit, current phase and issue date. Clicking any contract opens a page with more detailed info containing again the basic info plus deadline for bids, ISVZ number (if the contracts is sent to ISVZ), contract's kind and estimated price, description of what is the subject of that contract, info about the submitter and a contact address. This is all simple text marked with HTML (mostly HTML tables or unordered lists). Below the detailed info, there are usually couple of tables with links to attached documents, such as tender's documentation, additional messages or forms. As for the forms, these are again HTML documents, mostly with information either about what goes to ISVZ or about the winner, number of bids and contract's price. This structure remains the same for all the submitters and their E-ZAK pages.

One advantage is that the data are pretty well structured. And even if there is sometimes some information missing or new one appears, the structure is not changed and nothing is influenced. What I see as a shortage is that there is usually just textual description of the subject of a public contract, but the CPV codes[3] are missing. And if they are mentioned it is inside the text. A good point is publishing all the attachments. These do not go to ISVZ and so cannot be found anywhere else. It also seams that everything that has been once published stays in the system and is not removed or moved somewhere else. Thus, full procurement history for each submitter is kept. On the other hand, there are sometimes missing important forms about the results of public contracts. Important note is also that the data are spread over several web pages.

3.3 Problem Analysis

After selecting a data source, the application should scrape it for public contracts, triplify them and export as an RDF file. Users specify the serialization format of output file. And that is basically it. The triplification itself is conditioned by getting the data. That is the main problem - scraping the data from HTML pages so it can be processed, specifically exported as RDF triples. Since every public contract means accessing at least one new web page, there is a lot of network communication which may potentially consume quite a lot of time. However, there are not any specific performance requirements for the scraper. Once the contracts are scraped they can be transformed into an RDF graph. Finally, this graph can be serialized in different RDF formats. That will have impact at least on the size and human-readability of the resulting file. Since this is kind of

[2] `https://ezak.e-tenders.cz/contract_index.html`

[3] `http://simap.europa.eu/codes-and-nomenclatures/codes-cpv/codes-cpv_en.htm`

experimental application, we want to serialize the triples in all the basic formats (N3, RDF/XML, Turtle, N-Triples) in order to compare how suitable they are for public procurement.

Key point for the triplification process is an RDF vocabulary. OpenData.cz initiative developed an ontology for public procurement [48]. Following model is based on both parts (PC[4] and PCCZ[5]) of that ontology.

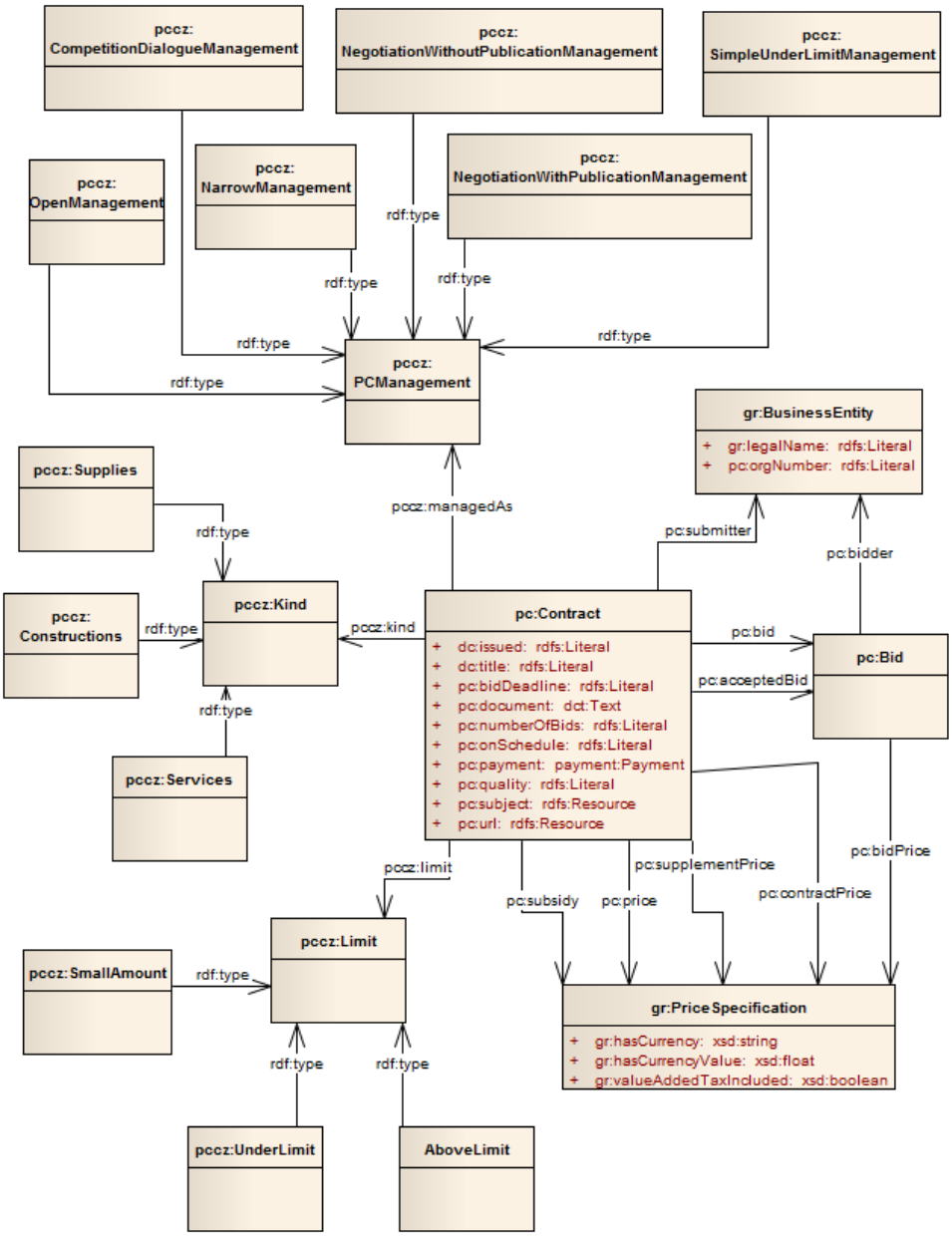


Figure 3.1: PC and PCCZ ontologies diagram

The diagram shows what attributes are held for each contract and how they are structured. The `pc:Contract` class represents a public contract. It has a set of basic properties describing the title, issue date, number of bids and others. Information about any price, such as contract price, supplement price or bid

[4] `http://purl.org/procurement`
[5] `http://purl.org/procurement/czech`

price, is represented as an instance of `gr:PriceSpecification` class. That one has three properties which specify the amount, currency and whether the tax is included. Any number of bids can be assigned to contract with `pc:bid`, however the winner should be in the `pc:acceptedBid` property. Any bid is an instance of `pc:Bid` class. Its price is represented by `gr:PriceSpecification` and the bidder, as well as the contract submitter, are instances of `gr:BusinessEntity` class which usually holds just organization name and number. So far, the model could be used for any public contracts, but there are also three properties specific for Czech procurement. Contract limit is expressed as an instance of one of the classes pointing to `pccz:Limit`. Same applies for `pccz:kind` and `pccz:managedAs` properties whose value is one of the classes of `pccz:Kind` or `pccz:PCManagement` type respectively.

3.4 Solution Proposal

As a solution proposal I will design an architecture for the application and describe individual components.

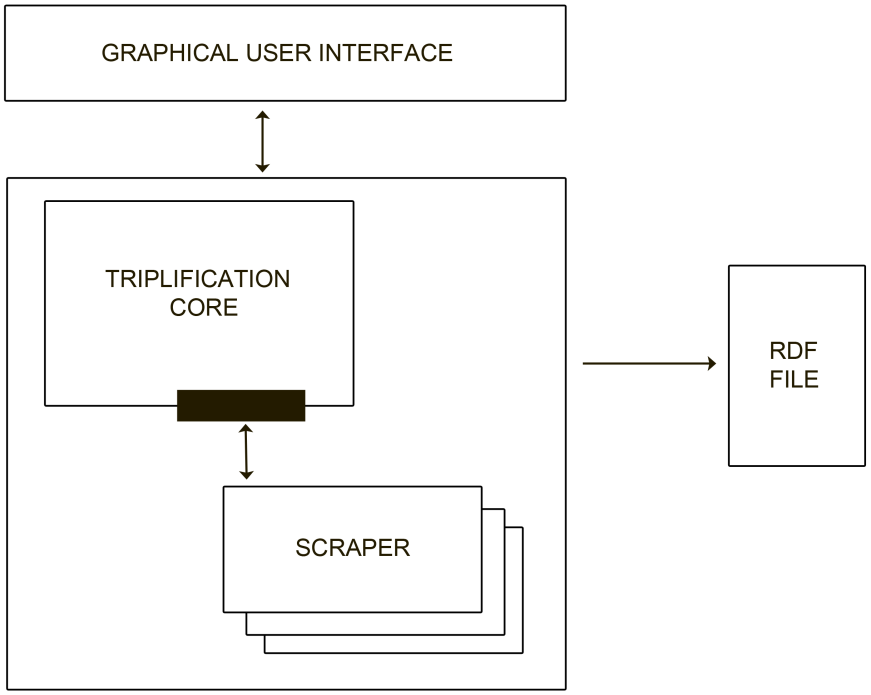


Figure 3.2: Suggested architecture diagram

Although I am scraping public contracts only from E-ZAK, I think the architecture of this application should allow possible later extensions for other systems administrating public procurement. Therefore, the triplication itself will be a core functionality of the application independent of the scraper.

The scraper will be realized as a plug-in implementing specified interface. It will be responsible for scraping all possible info about public procurement and returning the results to the triplification core.

The triplification core communicates with the scraper and calls its method to get scraped contracts. Its main purpose is to break the data returned from scraper down into triples and create an RDF graph. The public procurement RDF ontology [48] created by OpenData.cz will be used. This component is also responsible for the subsequent graph serialization.

The application will have a simple GUI where the user will be able to select what system with public procurement should be scraped and start the process. It will provide a table where the returned contracts will appear. Finally, there will be a selection of output format in which the RDF graph can be serialized. User will choose where to save the RDF file and how to name it.

Considering this solution proposal, appropriate technologies should be selected to make the implementation as simple as it gets. This basically consists of finding a simple solution for application's modularity (in other words, the ability to add plug-ins) and a well-documented, easy-to-use library for working with Resource Description Framework.

3.5 Implementation

Since this application makes no special demands on the programming language, I was choosing between C# and Java. Finally I went with Java, because it is platform independent and also has a wide support in the Semantic Web.

The modularity that enables other possible plug-ins to be added to the application is ensured by using the Service Provider Interface (SPI)[6]. That is a very basic method of making the application extensible. In fact, SPI is a set of well-defined public interfaces and abstract classes that the service provider (plug-in, in our case the scraper) must implement. Before the service provider is distributed as a JAR file, a configuration file must be created in `META-INF/services` folder. This file must be UTF-8 encoded and named with the fully qualified binary name of the service it implements. Inside the file, there should be fully qualified binary names of concrete implementations, one per line.[55]

I use Apache's Ant to build the JAR file for the scraper and it makes adding the configuration file as easy as adding the `service` directive. Here is the complete target for building the JAR:

```
<target name="makejar"
        depends="init, compile, compileezak"
        description="Make ezak.jar">
    <jar jarfile="${ezak.jar}">
<fileset dir="${bin}">
<include name="ezak/*"/>
</fileset>
<service type="logd.spi.PublicContractsScraper"
         provider="ezak.EzakScraper" />
</jar>
</target>
```

To use the scraper(service provider in general) in the application, the JAR must be added onto the class path. Then it can be found by the `ServiceLoader.load()`

[6] `http://en.wikipedia.org/wiki/Service_provider_interface`

method which is overloaded and can take a custom class loader as an argument. Here is the code from the application:

```
private Iterator<PublicContractsScraper> iterator;
private ServiceLoader<PublicContractsScraper> loader;
...
loader = ServiceLoader.load(PublicContractsScraper.class);
iterator = loader.iterator();
```

The `iterator()` method returns the iterator over all found plug-ins. At the meantime, there is just the E-ZAK plug-in.

The SPI is defined in `logd.spi.PublicContractsScraper` and contains only two methods: `getCZPublicContracts()` which returns `List<PublicContract-CZ>` with scraped public contracts, and `getSystemName()` which is used just to display the name of the data source in the application.

To represent a public contract in the memory I created the `PublicContractCZ` class in `logd.cz` package which is the subclass of `PublicContract` from `logd` package. This class was designed to hold info about the contract according to the RDF ontology for public procurement created by OpenData.cz. The scraper is implemented to parse even more information than that, but the code is commented out.

All the scraper classes are in the `ezak` package. `EzakScraper` is the main entry point implementing the `logd.spi.PublicContractsScraper` interface. Since there is many submitters using E-ZAK each having their own E-ZAK website, new thread is created for each submitter to scrape more sources in parallel. Each thread creates an instance of `ScreenScraper` class and calls its `getEzakContracts(String url)` method with an appropriate argument. So the scraping itself is implemented in `ScreenScraper` class.

The application has a simple graphical user interface implemented using the Swing toolkit. It is a good practise to execute time-consuming tasks on a different thread than Event Dispatch Thread that takes care of the GUI-related activities. Thus the scraping, which is a long running process, is executed in the SwingWorker Thread implemented as `ScrapeWorker` in `Application` class.

3.5.1 Scraping with *jsoup* Library

I used the *jsoup*[7] Java library for parsing HTML. It is an open source project providing a rich API for scraping HTML. It offers both, DOM methods and CSS-like selector syntax, and can parse even invalid and dirty HTML documents.

The basic procedure to scrape any HTML document is to parse it to a variable of `Document` type. *Jsoup* provides methods to parse a document from a `String`, to parse just a fragment of HTML document, to load and parse a document from a file or to fetch and parse it from a URL which is the one I was using in the `ScreenScraper` class:

```
Document doc = Jsoup.connect(String url).get();
```

At first, in the `ScreenScraper.scrape(String url)`, this method is used to parse a page with a list of contracts. Then it is called inside the `ScreenScra-`

[7] `http://jsoup.org/`

per.scrapeContract(String url) method to parse the main page for each contract. And the last use case is to parse some contract attachments, such as the form with results.

Once the document is parsed into a Document, *jsoup* offers many methods to extract or modify the data by combining two basic approaches - DOM-like methods and CSS-like selector syntax.

For cases when we know the structure of the document, there are many methods for navigating the DOM tree and getting element data. These are, for instance, methods to find elements by ID, tag, class, attribute, methods to find sibling elements, parent or children elements, methods to get attributes or text of elements. I used the DOM-like methods a lot, mainly to get the text content or attribute of concrete elements, but also to get all child elements. Here is an example from the source code:

```
name = row.child(0).text();
link = row.child(0).getElementsByTag("a").first();
href = link.attr("abs:href");
```

These lines of code show how contract name and link to the page with details are mined. Element row variable holds HTML for a row of table with basic info about contracts. The structure of such a row is:

```
<tr>
  <td>
    <a href="contract details url">
     "contract name"
    </a>
  </td>
  <td>"contract limit"</td>
  <td>"procurement phase"</td>
  <td>"issue date"</td>
</tr>
```

Thus, calling row.child(0) returns the first child element which is the first td. To get the "contract name", text() is called which returns element's text. To get the URL of the page with more details about contract, firstly the getElements-ByTag("a") is called which returns all the a elements and first() selects the first (in this case also the only one) of them. Next, the attr method is called with abs:href argument specifying we want the value of href attribute as an absolute URL.

The latter approach is implemented by select(String selector) method available in Document, Element and Elements. Its parameter is a CSS-like selector and the result is always of Elements type. There are many selectors, pseudoselectors and their combinations that allow writing powerful queries. Following code is at the beginning of ScreenScraper.scrape(String url) method.

```
Element hrefToNextSubpage =
              doc.select("a.nav:containsOwn(>)").first();
```

The selector says we want an a element with class attribute set to "nav" and, on top of that, it must directly contain ">" text. Because select(String selector)

returns `Elements`, the first result is returned with `first()` method. This is how the scraper gets link to next page with public contracts.

Another example can be selecting table rows with information about attached forms:

```
for (Element row : formsDiv.select("tr:gt(0)"))
...
```

The `formDiv` variable holds parsed `table` element. The selector returns `tr` elements whose index is greater then zero. In other words, it returns all rows except the first one which is the table header and does not contain any information for parsing.

Two more implementation notes about the scraper. The first one is that the default timeout for `Connection.get()` method is 3 seconds. Leaving that value, I was getting many exceptions and many HTML documents were not scraped. Therefore a custom timeout is set using the `Connection.timeout(int millis)` method.

```
private static final int TIMEOUT = 15000;
...
Document doc = Jsoup.connect(url).timeout(TIMEOUT).get();
```

The latter is the scraper does not implement parsing results information about other than the small amount public contracts. The reason is simple. Results in E-ZAK are posted as a separate form. And for under limit and above limit public contracts this is exactly the same form that goes to ISVZ. Fortunately, ISVZ was already triplified by Pavel Nohejl.

3.5.2 Triplification with *Jena* Framework

The core functionality of the application - creating an RDF graph from scraped data and serializing it in one of the RDF serialization formats - is built with *Jena*[8]. *Jena* is a semantic web framework for Java providing programmatic environment for RDF, RDFS, OWL and SPARQL. It is an open source project, therefore different developers have been able to code many enhancements for and on top of it. Not only that makes *Jena* framework widely used by the Semantic Web community.

First feature I would like to highlight is, that *Jena* comes with couple of wrapper classes for some basic RDF vocabularies like Dublin Core[9], RDF, RDF Schema, OWL, OWL2 and so on. Wrapper classes are a way of defining all resources and properties of an RDF vocabulary in one spot. That makes it easier to implement, maintain and use them [56]. Hence, I encapsulated the two vocabularies for public procurement (PC and PCCZ), and also those resources and properties from GOODRELATIONS[10] ontology they make extensive use of, in wrapper classes in `logd.vocabulary` package.

[8] `http://jena.sourceforge.net/`
[9] `http://dublincore.org/`
[10] `http://purl.org/goodrelations/v1`

```
public class PROCUREMENT
{
//URI for vocabulary elements
protected static final String URI = "http://purl.org/procurement#";

public static String getURI()
{
return URI;
}

//Definition of labels and objects
private static final String SUBBMITER = "submitter";
private static final String SUBJECT = "subject";
private static final String CONTRACT_CLASS = "Contract";
...

public static Property submitter;
public static Property subject;
public static Resource contractClass;
...

submitter = new PropertyImpl(URI, SUBBMITER);
subject = new PropertyImpl(URI, SUBJECT);
contractClass = new ResourceImpl(URI + CONTRACT_CLASS);
...
}
```

This extract from the code for `PROCUREMENT` class shows how two properties and one resource are encapsulated. At the be beginning, the base URI for the ontology is defined as a private field together with a getter. Next, I defined labels for the properties and resources. It does not matter how these variables are named, but, of course, their values must match the names in the vocabulary. This step is not necessary, the values could be also directly passed to the constructors for `PropertyImpl` and `ResourceImpl` classes. Following these are the declarations of public `Property` and `Resource` variables for each property or resource respectively. These can be later used anywhere, where the `PROCUREMENT` class is imported. The last step is to instantiate them. In case of properties, the constructor of `PropertyImpl` is called that takes two parameters - namespace URI and property name. Resources are instances of `ResourceImpl` class whose constructor takes the full qualified URI.

However the main purpose of *Jena* is to create and manipulate RDF graphs. Creating an empty RDF graph is as simple as calling `ModelFactory.createDefaultModel()` function. It returns a memory-based implementation of `Model` interface. Besides that, *Jena* contains also implementations for persistent models[11]. Once the model is created, resources and properties can be added. Resources and properties are represented by `Resource` and `Property` interfaces respectively. They are both associated with a concrete model and thus created by

[11]RDF graphs are called models in *Jena* and are represented by the Model interface

`Model.createResource` or `Model.createProperty` methods. The `Model.createResource` method is overloaded and can be parameterless (returns new blank node) or take one either `AnonId` parameter (returns new blank node with specified identifier) or `String` parameter (returns new resource with specified URI). The `Model.createProperty` method takes two `String` parameters to specify the namespace and local name parts of property URI. Besides resources and properties, we can also create literals using the `Model.createLiteral` method. While simple literals can be passed just as ordinary strings, any other untyped literals, such as the ones with language tag, should be created by this method. For typed literals, overloaded `Model.createTypedLiteral` method is provided.

To actually create statements, properties must be added to resources. Again, this is very simple in *Jena*. There are two overloaded methods to do so: `Resource.addProperty` and `Resource.addLiteral`. They usually take two arguments, first one is the `Property` and the latter is value of that property.

In the application, the triplification itself is implemented in the `RDFConverter` class. The principle is that the `export` method creates new empty RDF graph and then iterates through the list of `PublicContractCZ` that was passed as an argument from the scraper. For each instance it creates RDF triples for all non-null attributes and add these triples to the graph. The code below is an extract from the `export` method.

```
model = ModelFactory.createDefaultModel();
model.setNsPrefixes(nsPrefixesMap);

for (PublicContractCZ pc : contracts)
{
resource = model.createResource(pc.getUrl())
.addProperty(RDF.type, PROCUREMENT.contractClass);
addProperty(resource, DCTerms.title, pc.getTitle(), "cs");
...
}
...
```

First of all a new model is created. Next line causes that a `Map` with prefixes mapped to namespaces is added to the model. These prefixes are then used in the serialized file. Then the for cycle loops through all the scraped contracts passed in the `contracts` variable. At first, new resource is created with a URI stored in `PublicContractCZ.url`. Now the chain of `Resource.addProperty` methods should follow, but there is just one. That one assigns the resource an `RDF.type` property with `PROCUREMENT.contractsClass` value. That is a prefect example of using wrapper classes. However, the latter `addProperty` method in the code above is `RDFConverter.addProperty` method and so are the others. I came across a problem that it is not allowed to pass `null` value to the `Resource.addProperty` method. Consequently, to avoid writing a lot of null checks I refactored it into a separate method. This method performs the null check and possibly calls the `Resource.addProperty` :

```
public static void addProperty(Resource res, ...)
{
if(obj != null)
```

```
{
res.addProperty(...);
}
}
```

I omitted all the arguments but the first one, because this method is overloaded and same applies for all its modifications. The first argument is the resource I would have normally called the method on. In other words, the resource whose `addProperty` is called if the null check is false. Also note, that I was not using the `Resource.addLiteral` method at all, because the `Resource.addProperty` can be used for literals as well.

Once all the contracts are triplified and an RDF graph is created, they can be serialized. *Jena* has four built-in basic serialization formats:

- RDF/XML
- N-Triples
- Notation 3
- Turtle

To serialize the graph in RDF/XML format, there are actually two options: `''RDF/XML''` or `''RDF/XML-ABBREV''`. The latter produces more readable output by using the possible abbreviations in the syntax, but is not that efficient. Moreover, there are even four writers for Notation 3:

`"N3"` standard writer which chooses on the other three

`"N3-PP"` uses all the possible syntactic abbreviations

`"N3-PLAIN"` does not nest blank nodes, but groups properties for the same subject

`"N3-TRIPLE"` writes one statement per line

To serialize a model, its `write` method is one option. It takes up to three arguments to specify the `OutputStream` or `Writer`, serialization format and base URI. Another option is to use an `RDFWriter` returned by the `Model.getWriter` method with one optional parameter specifying the serialization language (format). `RDFWriter` is configurable and allows to customize the output a little bit. However, this only applies for `''RDF/XML''`, `''RDF/XML-ABBREV''` and `''N3''`. In the `RDFConverter.export` method I implemented the serialization using the `RDFWriter`, because I wanted to include the XML declaration for RDF/XML files. For other formats it is not different from using the `Model.write` method.

```
OutputStream outputStream = null;
if(!outputFile.exists())
{
outputFile = new File(outputFile.getPath());
}
try
```

```
{
RDFWriter writer = model.getWriter(outputFormat);
if(outputFormat.equals("RDF/XML-ABBREV"))
{
writer.setProperty("showXmlDeclaration", true);
}
outputStream = new FileOutputStream(outputFile);
writer.write(model, outputStream, null);
}
```

This code snippet shows how the graph is serialized at the end of `RDFConverter.export` method. The `File outputFile` parameter holds information about the output file. If it does not exist, a new one is created, otherwise the existing one will be overwritten. Then `RDFWriter` is created for the format chosen by user. If this is RDF/XML format, `showXmlDeclaration` property is set to `true` to write the XML declaration to the output stream. The `''RDF/XML-ABBREV''` writer for RDF/XML serialization is used for better readability. Finally, `FileOutputStream` is instantiated for the output file and the model is written to it. The last parameter (`null`) means that no base URI is specified.

Reading an RDF file into a model is very similar to writing RDF. Again, there are two options - either `Model.read` overloaded method or `RDFReader` for configurable reading.

3.6 Testing

I was testing mainly the functionality of the application and the results it was giving. The performance is not that important.

As for the scraper, I focused mainly on whether it works correctly for all possible structure scenarios of HTML documents. The individual parts responsible for parsing different sections of the HTML document, were being tested during the development. Basically all the tests were realized by running the scraper on the real data, observing the results and comparing them to original documents.

One problem that occurred during testing was that many documents were not scraped because of the low default timeout for the `Connection.get()` method as described in the previous section. Of course, it depends on the speed of the Internet connection, but I set the timeout to 15 seconds which should be enough even for slow connections.

The triplification part was tested on data returned by the scraper. The procedure was very similar. Again, it was a manual control of the results. In addition, I used the RDF validator[12] to check the validity of serialized RDF graph.

At the end, the GUI and the entire application were tested simply by running it and checking the behaviour and outputs.

[12] `http://www.w3.org/RDF/Validator/`

3.7 Discussion

To summarize it, I must say that E-ZAK is among the better public procurement administrators in the sense that it usually provides complete information about the contracts together with the appropriate documents and even old contracts stay in the system.

The built application allows to scrape the contracts published via E-ZAK and export them in one of four RDF serialization formats (RDF/XML, N3, Turtle, N-Triples). On top of that, the architecture allows adding other data sources scrapers (plug-ins) as long as they implement the specified interface. The application is far away from being perfect, but fulfilled its purpose - getting sample data. Next possible features could be scraping only contracts published in a specified time range and a possibility to clean the data before triplification.

Let me review the data I got. As for the quantity, about 860 contracts were scraped from E-ZAK. That means approximately 23,000 RDF triples. That is, in my opinion, a decent quantity and will be sufficient for further visualization. The results show that also the quality is pretty good. Except for three things I want to discuss.

One is that the subject is provided as a textual description, but the CPV codes are missing. Plus, I have not implemented the `pc:payment` property, because none of the systems provide this information and also the *OpenData.cz* tutorial does not mention it.

Another one is about information on the results of public contracts. As I mentioned in the section about implementation, the scraper only parses results about small amount public contracts (these are not in the ISVZ). However, they are very often missing. In addition, if available and the contract has more suppliers, there is no detail about how the contract price is divided between the suppliers.

The latter is that the contract might be sometimes split into a couple of parts. Unfortunately, the RDF ontology does not reflect it and also the scraper is ignoring this fact.

Besides that, some minor dirty data can appear, but there is usually no way to clean it but manually.

From the programmer's point of view, the most time-consuming and frustrating part was the scraper. Even though the data in E-ZAK have quite a good structure and are almost clean, there were several little differences in some HTML documents that had to be taken in account. I can definitely recommend the *jsoup* library which is very flexible and made the scraping easier. My other advice would be to analyze the data source carefully, before one starts to code the scraper. The better you analyze the data source, the better scraper you make and the cleaner data you get.

The triplification part took me much less time than the scraper. I think *Jena* is a good choice, too. The framework is very intuitive and easy to learn and use while providing great functionality. I would point out, that creating a wrapper class for the RDF vocabulary is a good first step. Not only it makes the code cleaner and easier to read, but also simplifies using the vocabulary. After I wrote the wrapper classes manually, I found out that *Jena* provides a command line utility called *schemagen*. This tool can automatically convert an ontology into a Java class.

At the end, I would like to mention, that during finishing this bachelor thesis, I ran the scraper to get new data. After that my IP address was blocked. E-ZAK team justified it by overloading their server and non standard behaviour. My request for unblocking the access was granted under following rules:

- accessing only one domain at a time
- maximum of 20 queries per minute

Unfortunately, this means that the scraper will have to be modified and should not be used in its current form.

4. Visualization

Once we have RDF data, the potential of possible applications, visualizations or mash-ups interpreting the data from different points of view, in different contexts, linked to other data and adding new value to them is enormous. In this chapter I will use the data I scraped and serialized in RDF, link them to another RDF data source and visualize the results. After the chapter about triplification, this is the latter step to provide an experimental application using RDF and Linked Data, more precisely, Linked Government Data.

4.1 Description

The visualization will be realized as a Web application and focused on small amount public contracts suppliers. In more detail, user will choose from a set of submitters using E-ZAK. The application will show a list of suppliers of small amount public contracts for selected submitters. In the next step, user specifies what suppliers he is interested in and finally he will be shown a map with corresponding table and a tree map. The map will show all the public contracts other than small amount, that the suppliers won and the table will provide more info and links to corresponding contracts. Besides that, there will be a tree map visualizing how much money the suppliers earned from public contracts and how this amount is split into individual contracts (submitters).

4.2 Analysis

The visualization has two steps. At first, the users sets the visualization options. That means that they choose what submitters they want to find suppliers for. The suppliers will be shown and users can select which ones they are interested in. These are small amount public contracts suppliers. Once the filters are set, the suppliers can be visualized.

There should be two problems visualized. First, the users want to see all other than small amount public contracts that the suppliers won. They want a map with these contracts, but also some basic info. Another requirement is, that there should be a possibility to get detailed information about any procurement on the map. The second visualization should focus on money. Users want to compare the suppliers according to how much money the earned from public contracts. Not only that, they also want to see, how the amount for each supplier is divided into individual contracts. In other words, how much money a concrete supplier had from what submitter.

As for the data sources, I will use the RDF data scraped from E-ZAK which contain also information about small amount public contracts and their suppliers. To get data about all other contracts, I will mash my data with ISVZ RDF data source triplified by Pavel Nohejl. Both data sets use the Public Procurement Ontology created by OpenData.cz. The ISVZ data is extended with information about organizations from ARES[1].

[1] `http://wwwinfo.mfcr.cz/ares/ares.html.cz`

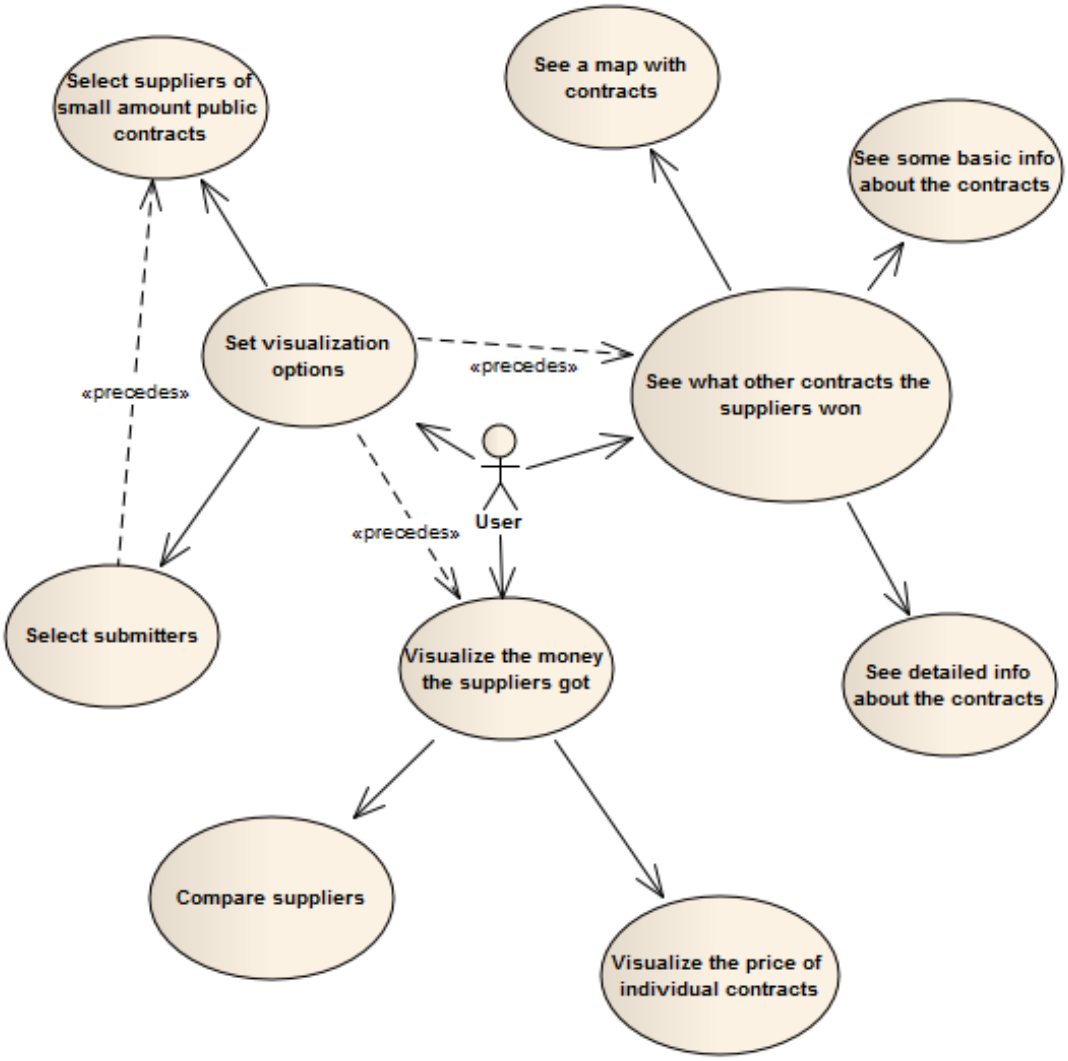


Figure 4.1: Visualization Use Case Diagram

4.3 Solution Proposal

The application will have three tiers - presentation, logic and data. The presentation tier is the communication point between the user and the application. It will get user's selections and display results and visualizations. This tier is also called client side.

The logic tier is probably the most important one. It will implement methods to process user's input and get data from RDF data sources to be passed to presentation tier. Not only it will query the RDF data set with E-ZAK contracts, but the logic tier will also communicate with the SPARQL endpoint through which the data about public contracts from ISVZ are queried. Querying the SPARQL endpoint will be done remotely using the SPARQL remote access protocol with HTTP. The results will be returned to presentation layer to be visualized.

The data tier is a tier that encapsulates the data sources.

4.4 Implementation

The application is implemented using Java Server Pages (JSP) technology[2] for generating dynamic pages. That allows to use Jena framework for querying RDF datasets. The data are visualized using the Google Visualization API[3].

The main application logic is inside the `EzakBean` Java class. When the user submits the form with submitters, the organization numbers for selected submitters are stored in `submitters` variable and `suppliersForSubmitters()` method is called. This method reads the RDF file with data from E-ZAK into an RDF graph. Then it queries that graph for suppliers of small amount public contracts for all submitters in `submitters` variable. The method returns a `ResultSet` containing submitter's organization number, supplier's organization number and supplier's name for every found supplier.

In the next step, when user submits another form, now with suppliers, the process is very similar. The organization numbers for selected suppliers are stored in `suppliers` variable and `suppliersContracts()` is invoked. This method creates a query to find all contracts that any of the suppliers in `suppliers` won and returns more info about those contracts, such as supplier's name, supplier's organization number, submitter's name, submitter's organization number, contract price and currency, submitter's latitude, longitude, locality and street and URL of that contract. Once the query is created, it connects to the SPARQL endpoint with contracts from ISVZ, executes the query and returns the results as a `ResultSet`.

```
QueryExecution qexec =
QueryExecutionFactory.sparqlService(
    "http://gd.projekty.ms.mff.cuni.cz:5000/
      openrdf-workbench/repositories/isvz_ares/query", query);
return qexec.execSelect();
```

This `ResultSet` is then converted to rows of `google.visualization.DataTable data` inside `drawVisualization()` JavaScript function. For each visualization control there is a `google.visualization.DataView` created to get only those columns from `data` that are necessary for that control.

One more important implementation note is on how the table and the map are connected. This is implemented by calling `google.visualization.events.addListener` function to register event handlers for `select` event for both charts. The function to be called when the event is fired gets the selection for the chart whose event has been fired and sets it as a selection of the other chart. In other words, if the map is clicked, selected contract becomes selected also in the table and vice versa.

Last thing I want to mention is about the tree map. For the purposes of this chart, the data are grouped by submitter-supplier pairs. This is done in `google.visualization.data.group` function where all the contracts with same submitter and supplier are summed and a total price is stored.

[2] `http://www.oracle.com/technetwork/java/javaee/jsp/index.html`

[3] `http://code.google.com/apis/chart/`

4.5 Testing

During the development, when testing the functionality of the application, I was I automated the process using *SeleniumHQ*[4] tool. It can be used as a Firefox plug-in that allows to record a test simply by interacting with the browser and then to replay it. So I was able to, for instance, record the process of setting visualization filters and did not have to go through it manually when testing the visualization charts.

I also ran couple of load tests to see how the application is responding under stress. I used a free Web debugging proxy *Fiddler*[5] with *StresStimulus*[6] extension which together offer an easy way how to record, configure and perform a test plan.

I ran multiple tests with different number of virtual users and various scenarios. The test plan was always the same: *main page – app.jsp – submitters.jsp – select all submitters and submit – suppliers.jsp – select all suppliers and submit – visualization.jsp*. I was mainly interested in number of errors and the response time.

Unfortunately, the *visualization.jsp* page has a high percentage of errors. I found out these are caused by `java.net.SocketTimeoutException` thrown while fetching the SPARQL endpoint. The problem is, that while it was not a problem in the scraper to change the default timeout value for getting HTML documents, *Jena* does not yet implement this option for connecting to remote SPARQL service. Following table shows the results for tests running for 3 minutes with constant number of virtual users:

Virtual Users	**Errors %**
5	31.4
10	39.7
20	66
50	96.5
100	98.5

The average response time is another attribute I was looking at while testing. It is said that 10 seconds si about the time when the user starts thinking something is wrong a leaves the page.

After running several load tests with both, constant and changing number of virtual users, I would say that the limit for a reasonable response time is about 150 virtual users. The 4.2 graph shows results of a test that starts with 1 user and simulates adding 10 more users every 5 seconds up to the number of 300 users. The browser type parameter was set to "Chrome".

4.6 Discussion

Created Web application shows how simple it is to use RDF data to create interesting visualizations and show the data in a way that everybody can understand

[4] `http://seleniumhq.org/`

[5] `http://www.fiddler2.com/fiddler2/`

[6] `http://stresstimulus.stimulustechnology.com/`

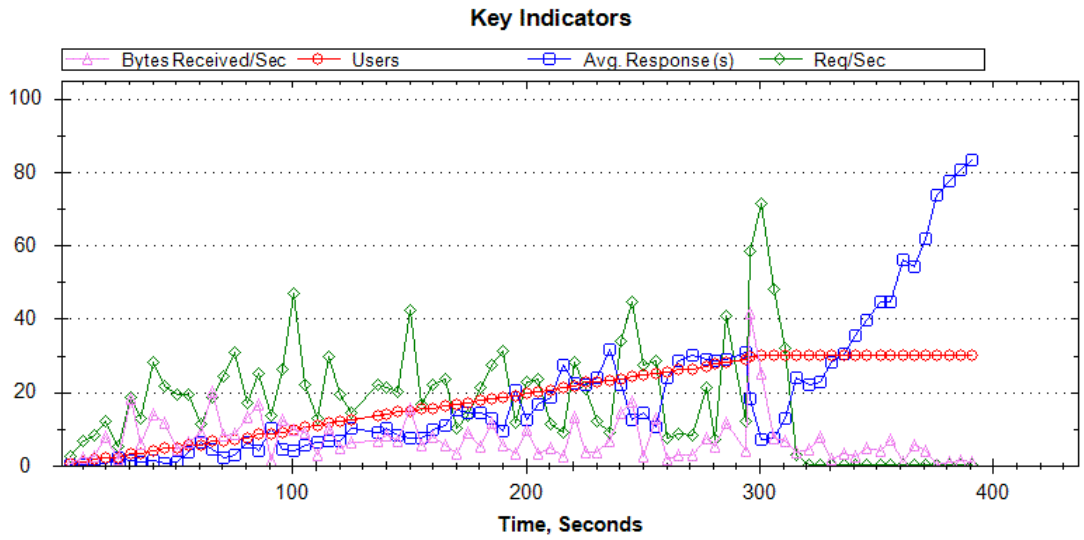


Figure 4.2: Load Test Graph

them. Thanks to the tree map, for instance, everybody can tell what supplier earned more money in the public procurement compared to the others, what supplier earned the most money or what suppliers earned pretty much the same. Everything at a glance. The map shows where the clients of those suppliers are from. The user can check what contracts the suppliers won in his region or in his town and see more details in the table.

However, the main purpose of this application was not to build a perfect visualization with many features and outstanding graphics. The main purpose was to show that using RDF data this can be done with just a little effort, in a few lines of code.

I also want to discuss some problems that occurred. First, Jena does not yet implement the `QueryExecution.setTimeout` method and so sometimes the default timeout has been exceeded when querying remote SPARQL endpoint.

Another point worth mentioning is that `QueryExecutionFactory.sparqlService` method uses HTTP GET request which has limited length. Even though the HTTP POST should be used automatically if the maximum length is exceeded, it is possible to force the POST method with `com.hp.hpl.jena.sparql.engine.http.HttpQuery.setForcePOST()`.

A typical problem that might influence the results are dirty data. According to the author of ISVZ RDF dataset, about 1/4 of organization numbers are missing. Another example of dirty data can be the latitude and longitude. I do not think there is a solution but cleaning and completing the data manually.

Many bugs appeared for Google Visualization API. This API is still under development and has many leaks. As for the map, I would probably use Google Maps API next time. That one allows a lot of customizing. With Google Visualization API I was not even able to set a tooltip text for the markers on the map. Another problem is, that it only displays a maximum of 400 items. Also the tree map is sometimes not working properly. Even though the underlying data, which I checked, are correct, the tree map very often shows the same submitter more than once. The good thing about Google Visualization API is that multiple charts can share the same data a there is a lot of nice charts that can be used

very easily.

Conclusion

Open Government Data initiatives have been pushing governments all over the world to publish all their data on the Internet. Government Linked Data offers a way how to do that. Just publishing the data is not enough, because different formats do not allow other reuse. Government Linked Data means publishing data in RDF a linking them to other data. Although the linking itself can be done by anybody else, it is essential that the government uses RDF format.

At the beginning of this thesis I introduced the Resource Description Framework and associated technologies to provide theoretical background. Then I focused on public procurement on the Internet in the Czech Republic, pointed out the problems and showed how RDF and Linked Data could be used as a solution. To provide some real examples, I implemented a simple application to scrape and triplify public procurement. To meet the aim of working with Linked Data, I developed a web application mashing the scraped and triplified data with a Linked Data data source and visualizing the results. That being said, I have achieved the goals set at the beginning of the thesis.

To summarize the results of the thesis, current situation of public procurement is very troublesome. Public contracts are anything but transparent and nobody really knows how much money is spent on them. There are many public procurement administrators with various electronic systems that the submitters use. As a result, public contracts are spread among many places on the Web and published in different forms. Many details are missing, not all contracts are posted on the Internet and those that are, do sometimes disappear. Reusing the data costs a lot of time and effort as I experienced during the scraper development.

The first step to change this is to publish all data on the Internet. However, this step should be closely tied with a second step. That is publishing the data in RDF format. The third step could then be using Linked Data. The advantages for both sides are worth it. OpenData.cz initiative has developed an RDF ontology for public procurement together with a tutorial on marking existing HTML pages with RDFa. That is perfect solution for those who have already posted their procurement on the Internet. Using RDF would not only unify the way of posting public contracts online, but also allow the data to be used for other purposes than originally meant by the publisher. It is also easy for the developers to reuse the data in their applications, link them to other data and show interesting mashups. Consequently the data become more understandable even for people with no economic training. From the government's point of view, Linked Data offers a benefit of publishing the data responsibly and increases the transparency as well as corruption detection. Moreover, the consumers are adding new value to data which can lead to economical growth as proved by UK government [46].

For those, who are eager to start scraping and triplifying government data, I would like to summarize my experience in just a couple of remarks:

- Learn as much about the data source as possible. It is not a bad idea to observe it for some time to find out what happens with old data and whether already published data can change or not.
- If no information about the query limit is provided, contact the administrator and tell them about your purposes before you start overloading the

server with the scraper. You can avoid blocking your IP address as well as subsequent code changes.

- If using an existing ontology, study it carefully to make sure you use the classes or properties correctly. Looking at the names is definitely not enough.

This thesis could be further extended in two ways. One is expansion of the analysis from public procurement to government data in the Czech Republic in general. The latter is design and implementation of specialized RDF tools for public procurement submitters, by extension, government data publishers.

Bibliography

[1] Skuhrovec, J. *Kolik se utratí za veřejné zakázky? To u nás nikdo neví.* [online]. 15.02.2011 [cit. 10-07-2011]. Available at: `http://blog.aktualne.centrum.cz/blogy/jiri-skuhrovec.php?itemid=12218`

[2] Klyne, G., Carroll, J. *Resource Description Framework (RDF): Concepts and Abstract Syntax - W3C Recommendation* [online]. [cit. 12-07-2011]. Available at: `http://www.w3.org/TR/rdf-concepts/`

[3] Manola, F., Miller, E. *RDF Primer - W3C Recommnedation* [online]. [cit. 12-07-2011]. Available at: `http://www.w3.org/TR/rdf-primer/`

[4] *Resource Description Framework - Wikipedia* [online]. [cit. 12-07-2011]. Available at: `http://en.wikipedia.org/wiki/Resource_Description_Framework`

[5] Tauberer, J. *What is RDF and what is it good for?* [online]. [cit. 10-07-2011]. Available at: `http://www.rdfabout.com/intro/?`

[6] Beckett, D. *RDF/XML Syntax Specification (Revised) - W3C Recommendation* [online]. [cit. 12-07-2011] Available at: `http://www.w3.org/TR/rdf-syntax-grammar/`

[7] Zambonini, D. *The difference between XML and RDF* [online]. [cit. 20-07-2011]. Available at: `http://www.oreillynet.com/xml/blog/2005/09/the_difference_between_xml_and.html`

[8] Berners-Lee, T. *Notation 3* [online]. [cit. 11-07-2011]. Available at `http://www.w3.org/DesignIssues/Notation3.html`

[9] Beckett, D., Berners-Lee, T. *Turtle - Terse RDF Triple Language - W3C Team Submission* [online]. [cit. 23-07-2011]. Available at: `http://www.w3.org/TeamSubmission/turtle/`

[10] *Turtle(syntax) - Wikipedia* [online]. [cit. 23-07-2011]. Available at: `http://en.wikipedia.org/wiki/Turtle_(syntax)`

[11] Grant, J. Beckett, D. *RDF Test Cases - W3C Recommendation* [online]. [cit. 23-7-1011]. Available at: `http://www.w3.org/TR/rdf-testcases/#ntriples`

[12] *N-Tripels - Wikipedia* [online]. [cit. 23-07-2011]. Available at: `http://en.wikipedia.org/wiki/N-Triples`

[13] Carroll, J., Stickler, P. *RDF Triples in XML* [online]. [cit. 23-07-2011]. Available at: `http://sw.nokia.com/trix/trix.html`

[14] *TriX(syntax) - Wikipedia* [online]. [cit. 23-07-2011]. Available at: `http://en.wikipedia.org/wiki/TriX_(syntax)`

[15] Bizer, Ch., Cyganiak, R. *The TriG Syntax* [online]. [cit. 23-07-2011]. Available at: `http://www4.wiwiss.fu-berlin.de/bizer/TriG/`

[16] *TriG(syntax) - Wikipedia* [online]. [cit. 23-07-2011]. Available at: http://en.wikipedia.org/wiki/TriG_(syntax)

[17] Brickley, D., Guha, R.V. *RDF Vocabulary Description Language 1.0: RDF Schema - W3C Recommendation* [online]. [cit. 24-07-2011]. Available at: http://www.w3.org/TR/rdf-schema/

[18] *RDF Schema - Wikipedia* [online]. [cit. 24-07-2011]. Available at: http://en.wikipedia.org/wiki/RDF_schema

[19] McGuinness, D.L., van Harmelen, F. *WOL Web Ontology Language Overview - W3C Recommendation*

[20] *OWL 2 Web Ontology Language Document Overview - W3C Recommendation* [online]. [cit. 24-07-2011]. Available at: http://www.w3.org/TR/owl2-overview/

[21] Haslhofer, B., Momeni, E., Schandl, B., Zander, S. *Europeana RDF Store Report - The Results of qualitative and quantitative study of existing RDF stores in te context of Europeana* [online]. [cit. 20-07-2011]. Available at: http://www.europeanaconnect.eu/documents/europeana_ts_report.pdf

[22] Hertle, A., Broekstra, J., Stuckenschmidt, H. *RDF Storage and Retrieval Systems* [online]. [cit. 21-07-2011]. Available at: http://ki.informatik.uni-mannheim.de/fileadmin/publication/Hertel08RDFStorage.pdf

[23] Bönström, V., Hinze, A., Schweppe, H. *Storing RDF as a Graph* [online]. [cit. 21-07-2011]. Available at: http://www3.ntu.edu.sg/home/bshe/graphStorage.pdf

[24] *Triplestore - Wikipedia* [online]. [cit. 20-07-2011]. Available at: http://en.wikipedia.org/wiki/Triplestore

[25] Prud'hommeaux, E., Seaborne, A. *SPARQL Query Language for RDF - W3C Recommendation* [online]. [cit. 21-07-2011]. Available at: http://www.w3.org/TR/rdf-sparql-query/

[26] Clark, K.G., Feigenbaum, L., Torres, E. *SPRQL Protocol for RDF - W3C Recommendation* [online]. [cit. 21-07-2011]. Available at: http://www.w3.org/TR/rdf-sparql-protocol/

[27] Beckett, D., Broekstra, J. *SPARQL Query Results XML Format - W3C Recommendation* [online]. [cit. 21-07-2011]. Available at: http://www.w3.org/TR/rdf-sparql-XMLres/

[28] *W3C Opens Data on the Web with SPARQL - W3C* [online]. [cit. 21-07-2011]. Available at: http://www.w3.org/2007/12/sparql-pressrelease

[29] Seaborne, A., Manjunath, G., Bizer, Ch., Breslin, J., Das, S., Davis, I., Harris, S., Idehen, K., Corby, O., Kjernsmo, K., Nowack, B.

SPARQL Update - A language for updating RDF graphs - W3C Member Sumbission [online]. [cit. 21-07-2011]. Available at: http://www.w3.org/Submission/SPARQL-Update/

[30] *SRARQL Tutorial* [online]. [cit. 21-07-2011]. Available at: http://openjena.org/ARQ/Tutorial/index.html

[31] McCarthy, P. *Search RDF data with SPARQL* [online]. [cit. 21-07-2011]. Available at: http://www.ibm.com/developerworks/xml/library/j-sparql/

[32] *SPARQL - Wikipedia* [online]. [cit. 21-07-2011]. Available at: http://en.wikipedia.org/wiki/SPARQL

[33] Harris, S., Seaborne, A., Prud'hommeaux, E. *SPARQL 1.1 Query Language - W3C Last Call Working Draft* [online]. [cit. 21-07-2011]. Available at: http://www.w3.org/TR/sparql11-query/

[34] Adida, B., Birbeck, M., McCarron, S., Pemberton, S. *RDFa in XHTML: Syntax and Processing - W3C Recommendataion* [online]. [cit. 14-07-2011]. Available at: http://www.w3.org/TR/rdfa-syntax/

[35] Adida, B., Birbeck, M. *RDFa Primer - W3C Working Group Note* [online]. [cit. 14-07-2011]. Available at: http://www.w3.org/TR/xhtml-rdfa-primer/

[36] Pemberton, S. *RDFa for HTML Authors - W3C* [online]. [cit. 14-07-2011]. Available at: http://www.w3.org/MarkUp/2009/rdfa-for-html-authors

[37] *RDFa - Wikipedia* [online]. [cit. 14-07-2011]. Available at: http://en.wikipedia.org/wiki/RDFa

[38] Adida, B., Birbeck, M., McCarron, S., Herman, I. *RDFa Core 1.1 - W3C Last Call Working Draft* [online]. [cit. 15-07-2011]. Available at: http://www.w3.org/TR/rdfa-core/

[39] McCarron, S. *XHTML+RDFa 1.1 - W3C Last Call Working Draft* [online]. [cit. 15-07-2011]. Available at: http://www.w3.org/TR/xhtml-rdfa/

[40] Cyganiak, R., Jentzsch, A. *The Linking Open Data cloud diagram* [online]. [cit. 15-07-2011]. Available at: http://lod-cloud.net/

[41] *Informační Systém o Veřejných Zakázkách* [online]. [cit. 30-05-2011]. Available at: http://www.isvzus.cz/usisvz/

[42] *Open Government Data* [online]. [cit. 05-06-2011]. Available at: http://opengovernmentdata.org/

[43] *Open Government Data. What is Open Government Data?* [online]. [cit. 05-06-2011]. Available at: http://opengovernmentdata.org/what/

[44] *Open Government Data. Film* [online]. [cit. 26-07-2011]. Available at: http://opengovernmentdata.org/film/

[45] Berners-Lee, T. *Linked Data* [online]. 2009/06/18 [cit. 20-05-2011]. Available at: http://www.w3.org/DesignIssues/LinkedData.html

[46] Sheridan, J., Tennison, J. *Linking UK Government Data* [online]. [cit. 25-07-2011]. Available at: http://events.linkeddata.org/ldow2010/papers/ldow2010_paper14.pdf

[47] *Opendata.cz - Transparent data infrastructure* [online]. [cit. 20-05-2011]. Available at: http://opendata.cz/

[48] Klímek, J., Mynarz, J., Nečaský, M., Petrák, J. *Recept pro vyznačení veřejné zakázky v HTML stránce pomocí jazyka RDFa* [online]. [cit. 05-07-2011]. Available at: http://opendata.cz/vocabulary/procurement

[49] *ScraperWiki* [online]. [cit. 14-07-2011]. Available at: http://scraperwiki.com/

[50] Bizer, Ch. *The D2RQ Plattform - Treating Non-RDF Databases as Virtual RDF Graphs* [online]. [cit. 28-07-2011]. Available at: http://www4.wiwiss.fu-berlin.de/bizer/d2rq/

[51] *Triplify* [online]. [cit. 28-07-2011]. Available at: http://triplify.org/

[52] Bizer, Ch., Heath, T., Berners-Lee, T. *Linked Data - The Story So Far* [online]. [cit. 23-06-2011]. Available at: http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf

[53] *Semantic Web - Triplify* [online]. [cit. 28-07-2011]. Available at: http://semanticweb.org/wiki/Triplify

[54] *W3C Wiki - ConverterToRdf* [online]. [cit. 28-07-2011]. Available at: http://www.w3.org/wiki/ConverterToRdf

[55] O'Conner, J. *Creating Extensible Applications With the Java Platform* [online]. [cit. 13-06-2011]. Available at: http://java.sun.com/developer/technicalArticles/javase/extensible/index.html

[56] Powers, S. *Practical RDF* [online]. Sebastopol, CA, USA: O'Reilly Media, 2003 [cit. 15-06-2011]. Chapter 8: Jena: RDF in Java. Available at: http://oreilly.com/catalog/pracrdf/chapter/ch08.pdf. ISBN 978-0-596-10355-2.

Attachments

A. CD Content

This thesis is accompanied by a CD-ROM containing the electronic version of the text, source and executable files. The directory structure is as follows:

- documents - text of this bachelor thesis in pdf format, programming documentation for scraper and triplification
- scraperTriplifier - executable and source code files for the scraper and triplification
- webApp - source code files for Web application visualizing public contracts

B. User Documentation For Scraper And Triplification

B.1 Graphical User Interface

The application has a simple and intuitive GUI. Following screenshot shows the GUI when the application is started.

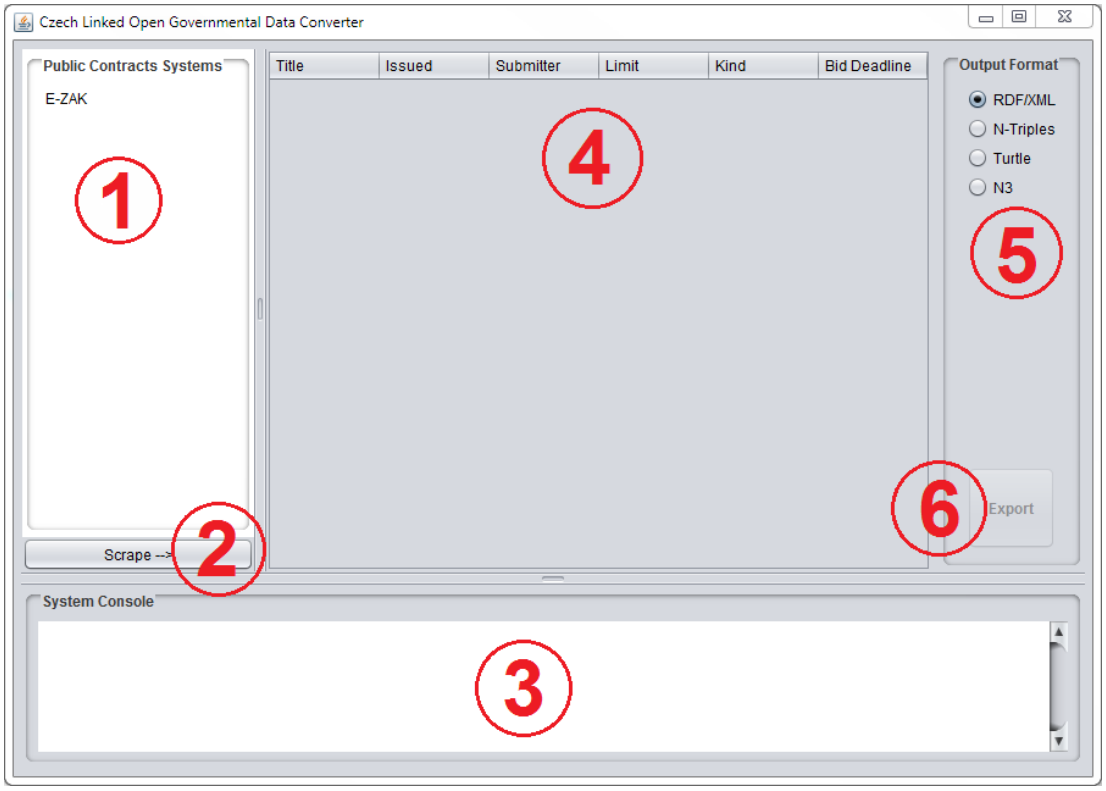


Figure B.1: Graphical user interface for the scraper and triplification application

1 - Panel with a list of systems that might be scraped. In other words, list of loaded plug-ins.

2 - Button to start scraping. During the process, there is a progress bar.

3 - Text area providing info about the scraping process, such as what contracts are being scraped or what errors occurred

4 - Table with basic info about scraped contracts

5 - Selection of serialization format for output file

6 - Button to export scraped contracts in selected format

Following steps describe the process of scraping and exporting public contracts in RDF:

1. Select the data source for public contracts (1).

2. Click the "Scrape" button (2). Scraping starts and information about the process is shown in the "System Console" area (3). During the process, there is a progress bar instead of the "Scrape" button informing the user that the scraper is running. The only way how to stop the process is to close the application! Once the scraper finishes, basic details about the scraped contracts are shown in the table (4) and can be sorted by any column (note that this will not influence the order of contracts in the output file).

3. Select one of the serialization formats (5).

4. Click the "Export" button (6). A "Save file" dialog will pop up. Select where to save the file and how to name it and click "OK".

5. You can repeat the process from the 3. step to export the same set of contracts in different formats.

C. Programming Documentation For Scraper And Triplification

Complete programming documentation for the whole application is on the accompanying CD-ROM in *documents* folder. It is an HTML file generated automatically from comment documentation by *Javadoc*[1]

[1] `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`

D. User Documentation For Visualization

The web application visualizing the data is available at `http://logdcz.appspot.com`. This application lets user select a set of available submitters, shows the suppliers of small amount public contracts for these and visualizes what other contracts the suppliers won and how much money they got for them.

To create and show new visualization, follow these steps:

1. Follow the "APP" link to `http://logdcz.appspot.com/app.jsp`.
2. Click the "START HERE" button or the "SUBMITTERS" tab.
3. Boxes with available submitters appear. Select the submitters to find small amount public contracts suppliers for and click the "SUBMIT" button.
4. The application returns list of found suppliers. Select the ones you want to visualize. To select all suppliers, click the "SELECT ALL". The button will change to "DESELECT ALL". To visualize the suppliers, submit your selection by clicking the "SUBMIT" button.
5. Finally the visualization is drawn. To change the visualization filters, start the whole process again.

The map shows places where the suppliers got some public contract. To be more precise, it shows locations of submitters of those contracts. The table next to it provides more info about the contracts - supplier's name, submitter's name, submitter's location, contract price and link to the contract page. Both the map and the table are connected, in the sense that when the user clicks either one, the contract is selected in the other as well. Unfortunately, the map can visualize maximum of 400 contracts. The user can sort data in the table by any column.

The tree map shows how the total amount of money from contracts is divided between individual suppliers. Click the left mouse button to see how the amount for a concrete supplier is split into individual submitters. Right-click to return back to the suppliers level.

The application also allows the user to download the data scraped from E-ZAK. On `http://logdcz.appspot.com/rdf_data.jsp` there are the data exposed in four different RDF serialization formats.

E. Programming Documentation For Visualization

The application was implemented using Java Server Pages technology combining HTML and Java programming language. Some client-side code was written in JavaScript.

E.1 Used Tools

Jena Open source Java framework for working with RDF, RDF Schema, OWL and SPARQL. I used this framework to query RDF file with contracts from E-ZAK as well as to query remote SPARQL endpoint.

Google Visualization API API for creating charts, tables and other visualizations and integrating them into websites. I used this API to create the map, table and the tree map.

jQuery Open source JavaScript library for easy HTML document navigation, event handling, animations or developing AJAX based applications.

Google Application Engine Engine for building and hosting web applications on Google's infrastructure. Can be used freely with limited resources. Offers many features, simple administration tools, fast deployment,... I use Google Application Engine for hosting the application.

E.2 Documentation

style.css CSS file for the entire application.

head.jsp Contains elements to be inserted into the `head` element of every page. Takes a `title` parameter to specify the page title.

header.html Simple HTML document containing elements for the logo and navigation menu with links to other pages. This document is included in all pages.

footer.html HTML document with copyright information and mail to the author. It is included at the end of every page.

tabs.html Defines the tabs as unordered list of links to appropriate sites inside the `tabs_wrapper` and `tabs_links divs`.

tab_style.jsp Implements the `style` element with background color for a link in `tabs_links div`. `Id` of the link is determined by `tab_href` parameter. This JSP file is included in *submitters.jps*, *suppliers.jsp* and *visualization.jsp*.

index.jsp Home page of the application. Includes `head.jsp` with "Home" string as an argument. Contains introduction text.

info.jsp This page includes `head.jsp` with "Info" argument. It serves as an info page about the application.

app.jsp Contains text with instructions on how to use the application. Passes "App" string to included `head.jsp` and together with `header.html` it includes also `tabs.html`.

submitters.jsp That is the first step to the visualization. The page contains a `form` with `checkboxes`. These are generated dynamically in a loop over the set of keys from `EzakBean.submittersMap`. That means there is one `checkbox` for each submitter and the value is set to the submitter's organization number. When the form is submitted the data are sent to *suppliers.jsp*.

jQuery is used to handle `checkboxes` checking and unchecking, because they are customized. The events and handlers are implemented inside the `$(document).ready()` function so they will load before the page contents are loaded. These are `click` events for `a` elements inside the list item with the `checkbox`. The first handler adds `selected` class to the list item element (`li`) and sets `checked` attribute of the `checkbox` inside this `li` element to "checked" if checking the `checkbox`. The latter handler does the opposite if a link to deselect the submitter is clicked. I implemented the customized checkboxes according to Aaron Weyenberg's tutorial[1].

suppliers.jsp At first an instance of `EzakBean` is created (if it does not yet exist) and its `submitters` attribute is set to contain all the submitters' organization numbers from the `request`. Then there is a `form` containing a `table`. To get all suppliers the `suppliersForSubmitters()` method is called on the instance of `EzakBean` class. Then the `while` cycle iterates through the returned `ResultSet` and adds a row for each `QuerySolution` to the table. Each row contains also a `checkbox` with the value set to supplier's organization number. When the `form` is submitted, the data is sent to *visualization.jsp*.

Selecting rows of the table, in other words checking the hidden `checkbox` in each row, is again implemented using the jQuery and adding `click` event for each row. jQuery's `toggle` function is used to implement selecting and deselecting all rows. The event is similar to the `click` event, but takes two functions as arguments and toggle between them every other click. First function adds `selected` class for the clicked row, sets the `checked` attribute of its `checkbox` to "checked" and changes the `src` attribute. The latter removes that class and that attribute and changes the `src` back.

visualization.jsp An existing instance of `EzakBean` is used and all the organization number's of suppliers from `request` are added to its `suppliers` attribute. If no instance exists, a new one is created.

[1] `http://aaronweyenberg.com/90/pretty-checkboxes-with-jquery`

- `drawVisualization()` - This function draws the map, table and tree map using Google Visualization API. An underlying `DataTable` is created by iterating through the results of `ezak.suppliersContracts()`. For each chart a separate `DataView` is created using only those columns from the basic `DataTable` that are relevant. For the map, only the columns with latitude and longitude are used. `TablePatternFormat` is used to create a link to contract web page and to merge price and currency columns for the purposes of the table chart. To connect the map with the table, `addListener` function is called for both charts to add event handler for their `select` event. For the tree map, the data is grouped by submitter-supplier pairs. So the price of all contracts that have same submitter and supplier is summed. Before drawing the tree map, extra row for each supplier plus one special row must be added to create the tree structure with one root. The extra rows are added dynamically from the `ResultSet` returned by `ezak.suppliersNames()`.

rdf_data.jsp Contains unordered list of links to triplified and serialized public contracts. from E-ZAK.

EzakBean.java Class implementing methods for querying RDF data.

- `Map<String, String> submittersMap` - Maps organization names of submitters to organization numbers.
- `String[] submitters` - Private array to hold submitters' organization numbers.
- `String[] getSubmitters()` - Returns `submitters`.
- `void setSubmitters(String[] submitters)` - Assigns the array in parameter to `submitters`.
- `ResultSet suppliersForSubmitters()` - Queries RDF file with contracts from E-ZAK using SPARQL and returns suppliers of small amount public contracts for submitters in `submitters` array.
- `String[] suppliers` - Private array to hold organization numbers of suppliers.
- `String[] getSuppliers()` - Returns `suppliers`.
- `void setSuppliers(String[] suppliers)` - Takes an array of `Strings` and assigns it to `suppliers` variable.
- `ResultSet suppliersContracts()` - Creates a SPARQL query to get information about all won contracts for suppliers in `suppliers` array, connects to remote SPARQL endpoint with data from ISVZ and ARES. Executes the query and returns a `ResultSet` with supplier's name and organization number, submitter's name and organization number, contract price and currency, submitter's latitude, longitude and locality and contract URL columns.

- `ResultSet suppliersNames()` - Returns names of suppliers in `suppliers` array.